2002-04

MR 2020 Computer Computations in Air-Ocean Sciences

Jordan, Mary S.

Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

MR 2020 Course Notes

MR 2020
COMPUTER COMPUTATIONS IN
AIR-OCEAN SCIENCES

by

Mary S. Jordan
Department of Meteorology
Naval Postgraduate School
Monterey, CA 93943
jordan@nps.navy.mil

April 2002

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION TO UNIX

A. UNIX SKILLS CHECKLIST

1. Log into your UNIX account.
2. Change password -- after choosing an password that is difficult to crack.
3. Log out of the UNIX workstation.
4. Security procedures for the MR and OC UNIX labs.
   - IDEA Lab (Root 123)
   - Graphics Lab (Spanagel 341)
5. Know the absolute path name for your home directory.
6. Understand UNIX case sensitivity.
7. Understand absolute and relative path names.
8. Understand the print work directory "pwd" command.
9. Make and remove directories ("mkdir" and "rmdir" commands).
10. Change directories with the "cd" command.
11. Know three ways to "cd" to your home directory from any directory.
12. Understand file name rules:
    - Case sensitive.
    - Understand permissible file names and invalid characters.
    - Root name and extension: rootname.extension
      (e.g. 20010111syn.gem)
13. Listing files and directories with "ls", "ls -l", 'ls -la", and "ls -C".
14. Understand and identify file permissions.
15. Understand "hidden" files.
16. Be able to identify directories in a listing.
17. Use of the wildcard *.
18. Copying files ("cp").

19. Removing (deleting) files ("rm").

20. Use of text editors "nedit" and "jot".

    - Open a file using text editor in a UNIX shell.
    - Understand the difference between opening a text editor from a menu and from a UNIX shell.
    - Understand the difference between a text editor and a word processor.
    - Understand why the "&" may be needed with nedit (e.g. nedit myfile.txt &).

21. Viewing the contents of files (without opening the files).

    - "more", "cat", and "tail" commands.

22. Understand the "history" command, "!!" and "!n".

23. Understand how to print a file from a UNIX shell (winterm) using "lp filename".

24. Understand the use of "Control-C" and "Control-D".

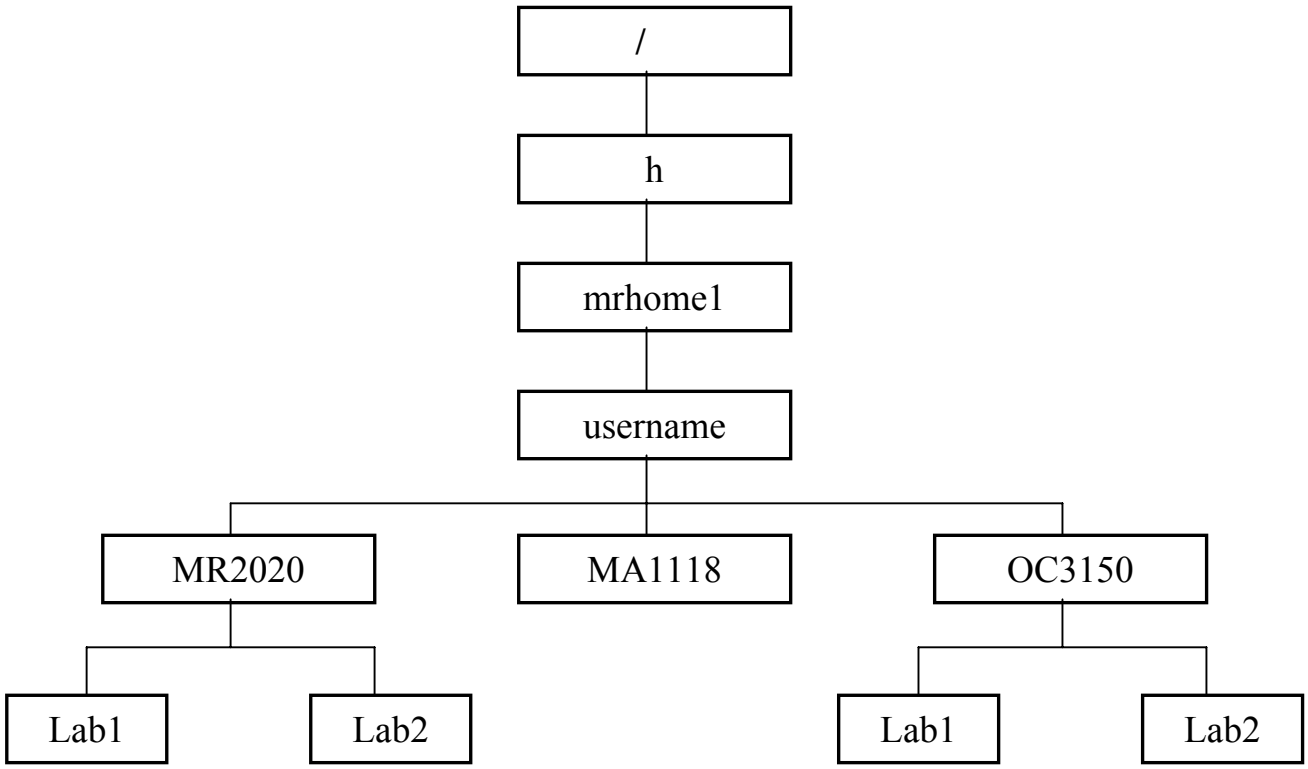
25. Use the "chmod" command to change file permissions.

26. Understand workstation procedures:

    - Never power off a workstation.
    - Never hit the reset button a workstation.
    - Never use a UNIX file icon menu as you would use a Microsoft Explorer window for copying files, etc.
    - Understand that other users may be remotely logged into you workstation and sharing the CPU.
    - At night and on weekends, power off only the monitor before leaving.

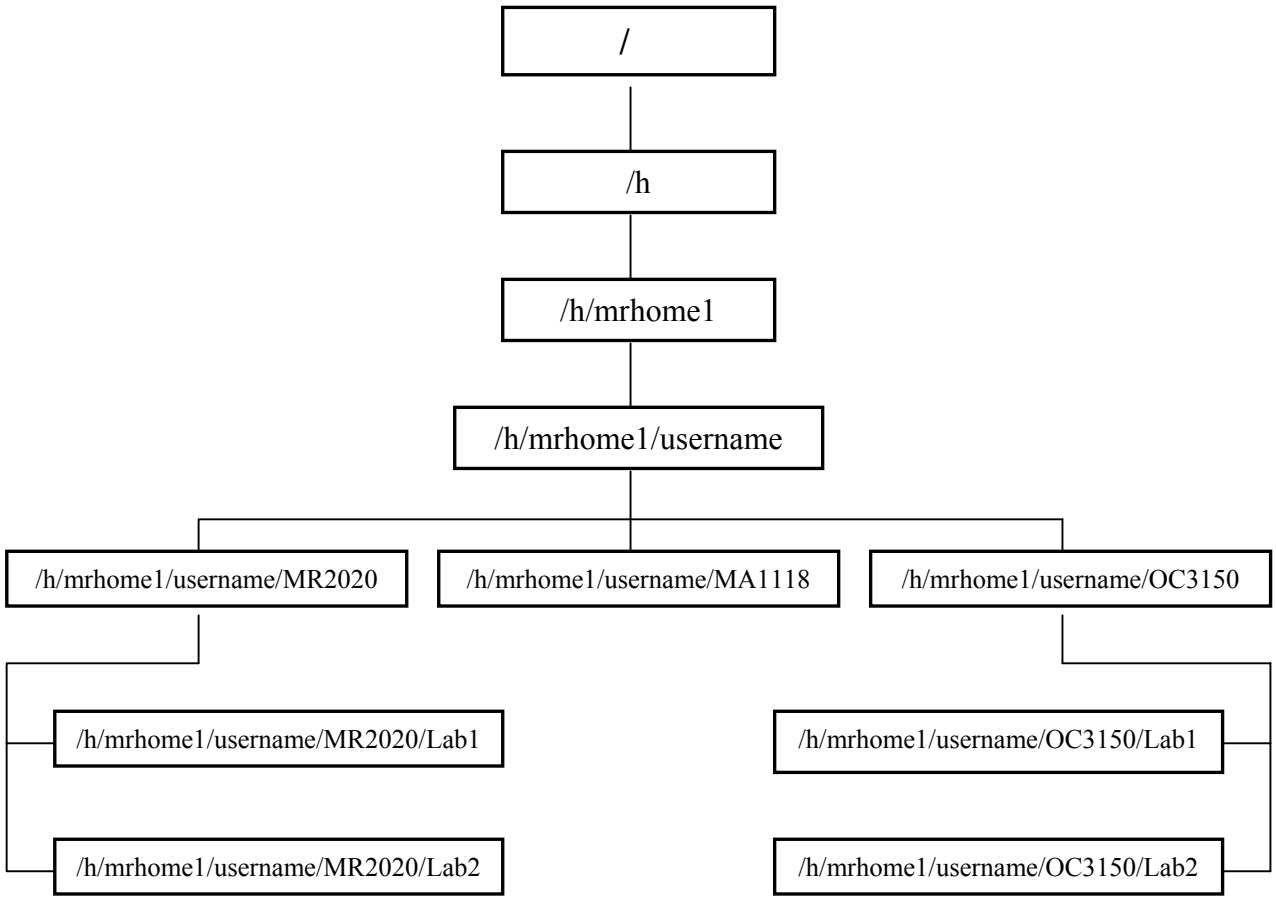
B. DIRECTORY STRUCURE WITH RELATIVE PATH NAMES

Relative Names of Directories



C. DIRECTORY STRUCURE WITH ABSOLUTE PATH NAMES

Absolute Path Names of Directories



D. COMMON UNIX COMMANDS

The following conventions are used in the UNIX command examples:

1. Any commands or other text, which a user must enter into the computer, will appear in boldface.
2. Every command line is to be completed with a press of the (RETURN) key.
3. All UNIX commands must be typed in **lower case**.
4. Some UNIX commands have options available, and the option is identified in the command with the dash, "**-**", e.g. *command –option*

Table 1. Common UNIX Commands

Command	Option	Description
man		The *on-line* **man**ual with a complete list of options available for each command. **man** command
pwd		Print working directory: tells absolute path from root to where you currently are in the file system.
cd		Change directory. Syntax: **cd** path/directory where *path/directory* is either an absolute or relative pathname to the desired directory.
	cd ~	Changes to the users' home directory.
	cd ..	Changes to the parent directory of the current directory (up one level).
mkdir		Make directory. Syntax: **mkdir** directoryname
rmdir		Remove directory. Syntax: **rmdir** directoryname Note: the directory must be empty to be removed.

ls		List files and directory names in the current directory in column format.
	ls -l	List files and directory names in long format, showing permissions, size, date last changed, etc.
	ls -la	Long listing, including *hidden files* (filenames starting with a period, ".")
>		Redirects output for the screen to a file.
	ls –l > *dir.txt*	Directory listing is stored in file *"dir.txt"*.
cp		Copies a file. Syntax: **cp** path/filename path/filename
rm		Removes a file. Syntax: **rm** *filename*
mv		Moves a file to a new path/name, with the original file deleted. **mv** *path/oldname path/newname* mv can be used to rename a file if the path is the same. *This can be a dangerous command if you make a typo!*
more		Displays the contents of a file, one screen-full at a time with a pause. To continue to the next screen, use the space bar. Syntax: **more** filename

tail		Displays the last 10 lines of a file to the screen. Syntax: **tail** filename
	tail -15	Displays last 15 lines.
head		Displays the first 10 lines of a file to the screen. Syntax: **head** filename
	head -15	Displays first 15 lines.
cat		Concatenates or joins files and prints to the screen.
	cat *filename*	Prints the contents of *filename* to the screen.
	cat f1 f2 > file3	Used with a ">", it can join or concatenate multiple files. Appends file "f2" to the **end** of file "f1" and saves it in a new file, *file3*. Original files are not changed.
lp		Print a file on the printer in the IDEA Lab and Graphics Lab. Syntax: **lp** *filename*
history		Lists previous commands by number.
	!n	Executes the n^{th} command.
	!17	Executes the 17^{th} command.
Ctrl-c		Emergency stop for a process. This stops print to the screen, infinite loops, etc. Works in both UNIX and MATLAB.
Ctrl-d		Kills the shell (winterm), which stops the execution of a process.
quota		Displays if user is *over* allotted disk space quota.
	quota –v	Displays user quota and amount of disk space used.

|		UNIX control character called a "pipe". This character is on the same key as " \ ". Used in some UNIX commands to direct the output of one command to be the input for the next command.
grep		A powerful search command. Can search a file and extract all lines containing a specified word or phrase. Output is to the screen.
	grep *smith file2*	Extracts all lines in file, *file2*, containing the word "smith". To search for a phrase, enclose the phrase in single quotes.
	ls –l | grep .m	Extracts all lines in the directory listing (ls –l) which contain the file extension ".m ".
ps		Lists all the processes currently running on the workstation (all windows, the desktop menu, the clock, programs, etc).
	ps -ef | grep username	Lists only the processes for *username*. The output lists the username, process identification number (PID), and process name: username 5544 /usr/bin/X11/xterm
kill		Stops the execution (kills) a specified process.
	kill 5544	Stops the execution of process 5544.

Table 2. Comparison of DOS and UNIX Commands.

Action	DOS Command	UNIX Command
Display list of files	**dir**	**ls**
Display file contents	**type**	**cat**
Display file contents one screen at a time	**type filename | more**	**more**
Copy file	**copy**	**cp**
Rename file	**rename**	**mv**
Delete file	**del**	**rm**
Create directory	**mkdir**	**mkdir**
Delete directory	**rmdir**	**rmdir**
Change working directory	**cd** or **chdir**	**cd**
Find string in file	**find**	**grep**
Compare files	**comp**	**diff**
Help	**help**	**man**
Print a file	**print**	**lp filename** (IDEA/Graphics Lab only)
Display the printer queue	**print**	**lpstat** (IDEA/Graphics Lab only)
Display date & time	**date** **time**	**date** (displays date & time)
Change file protection	**attrib**	**chmod**
Display free disk space	**chkdsk**	**df**

E. ABSOLUTE AND RELATIVE PATH NAMES

1. Definition: **Absolute Path** is the long path name, which can be used in any command.
    - For your account, the *absolute path* is similar to:

      /h/mrhome1/username
    - The *absolute path* for your mr2020 directory is similar to:

      /h/mrhome1/username/mr2020

1. Definition: **Relative Path** is the path relative to the directory in which you are working.
    - If you are in your home directory, the RELATIVE path to your mr2020 directory is:

      `mr2020`
    - The command, using relative path names, to change from /h/mrhome1/username is:

      `cd mr2020`
    - The command, using the absolute path, which works from any directory, is:

      `cd /h/mrhome1/username/mr2020`
2. Relative Directory Names
    - In each UNIX directory, two directory names are automatically made. You can only see these directory names with the **`ls -la`** command.
    - Example using directory: `/home/usr6/mr2020/lab1`

```
drwxr-xr-x  2 pasmith   user    512 Jan  4 13:21 .
drwxr-xr-x  6 pasmith   user   2560 Jan  4 13:21 ..
```

`"."`	means the current directory
`".."`	means the directory directly above the current directory (parent directory)
`cd ..`	move up one directory
`cd ../..`	move up two directory levels
`cd ../lab2`	move up one directory, then down into the directory lab2

3. Examples using the copy (cp) command

Example 1: In the same directory, copy file1.txt to file2.txt

Location: You are in the directory with file1.txt

```
cp file1.txt file2.txt
```

Note: All names are RELATIVE. Since you are in the same directory, there is no need to use absolute path names.

Example 2: Copy /home/usr6/mr2020/file1.txt
to /h/mrhome1/username/mr2020/file2.txt

a. Use ABSOLUTE path names (works from any directory)

```
cp /home/usr6/mr2020/file1.txt /h/mrhome1/username/mr2020/file2.txt
```

b. Mix ABSOLUTE and RELATIVE path names depending on the working directory location:

i. Location: /home/usr6/mr2020

```
cp  file1.txt  /h/mrhome1/username/mr2020/file2.txt
```

ii. Location: /h/mrhome1/username

```
cp /home/usr6/mr2020/file1.txt  mr2020/file2.txt
```

Example 3: Keeping the same file name (file1.txt), copy
/home/usr6/mr2020/file1.txt to directory
/h/mrhome1/username/mr2020

a. Use ABSOLUTE path names (works from any directory):

```
cp /home/usr6/mr2020/file1.txt /h/mrhome1/username/mr2020
```

Note: Unless told otherwise, UNIX copies to the SAME file name.

b. Mix ABSOLUTE and RELATIVE path names depending on your location:

i. Location: /home/usr6/mr2020

```
cp file1.txt /h/mrhome1/username/mr2020
```

ii. Location: /h/mrhome1/username/mr2020

```
cp /home/usr6/mr2020/file1.txt  .
```

Note: "." means the current directory

iii. Location: /h/mrhome1/username

```
cp /home/usr6/mr2020/file1.txt  mr2020
```

Example 4: Copying files within a relative directory structure defined as:

/h/mrhome1/username
/h/mrhome1/username/lab1
/h/mrhome1/username/lab2

a. Copy file1.txt up one directory:

```
cp file1.txt ..
```

Note: same command if file1.txt was in either subdirectory.

b. Copy file1.txt from subdirectory lab1 to lab2, keeping the same filename.

```
cp file1.txt ../lab2/file1.txt
or, cp file1.txt ../lab2/          (assumes same filename)
```

c. Copy file1.txt from subdirectory lab1 to lab2, but with a new filename, file2.txt

```
cp file1.txt ../lab2/file2.txt
```

F. USING THE "cd" COMMAND

1. Changing to your home directory.
    - The command ***cd***, without a pathname, returns to ***your*** home directory.

      cd

1. Using the tilde "~" character instead of a full pathname.
    - Use the ~ (tilde) character in place of the full pathname to your own or another user's home directory.
    - For example, to change your working directory to the home directory of the user "jksmith", enter the command:

      cd ~jksmith

    - Using the tilde in place of the full pathname is very useful in cases when you don't know the full pathname to the account home directory.

G. UNIX FILE PERMISSIONS

1. There are three types of access permissions: read, write and execute. There are different meanings for files and directories.

   r - read the file or directory

   w - write or delete the file or directory

   x - execute the file (i.e., run the program) or search the directory

2. Each of these permissions (read, write, execute) must be defined for three types of users:

   u - the user who owns the file (in your account, you own the files)

   g - members of the group to which the owner belongs

   o - all other users

3. The access permissions for all three user types are defined in one string of nine characters.

user	group	others
r w x	r w x	r w x

4. To view access permissions for files and directories, use the "**`ls -l`**" command.

```
drwx------  4 jordan   user        1024 Oct  5 20:05 Classes
drwxr-xr-x  2 jordan   user        1024 Jun 18 12:19 mr4416
-rw-r--r--  1 jordan   user        1161 Nov 21  2000 seaspace.txt
-rwxr--r--  1 jordan   user        1226 Aug  8  2000 tscaneagle
```

   - The "d" means that "Classes" and "mr4416" are directories.
   - "seaspace.txt" and "tscaneagle" are files, not directories, as indicated by the first dash "-".
   - The user permissions are in blue, group in red, and others in green.
   - For directory "Classes", only the user/owner "jordan" has permission to access.
   - For directory "mr4416", only the user/owner "jordan" has full permissions, and "group" and "other" users can access the directory, but cannot write or delete any file.
   - The file "seaspace.txt" may be read by "group" or "other" users, but not written, deleted, or executed.
   - The "tscaneagle" file is a program that the owner can execute (or run). Only the owner can execute this program, but "group" or "other" users can read the program code.

5. Directory rules:

   - For anyone other than the owner to access the files in that directory, the "group" or "other" permissions must be set to "r-x".
   - Separate permissions are defined for each file located in each directory.
   - The "mkdir" command, which defined a new directory, *automatically* sets the permissions as: **`drwxr-xr-x.`**

6. Security rule: do not allow "group" or "other" users write/delete "w" permission for any file or directory.

7. The permissions defined automatically to any newly created file are: **`-rw-r--r--`**.

8. Unlike DOS, where executable programs are automatically recognized by the file extension ".exe", UNIX has not automatic recognition of executable files. In UNIX, the file permission must be set to "x".
    - Use the "chmod" command to manually set a program to be a UNIX executable program.
    - Allowing someone other than the owner to execute a program can cause problems, especially when a program creates or overwrites files in the owner's account. Don't give "group" or "other" user execution permission.
    - If another user needs to share this program, the other user can copy the program to the other account, set "x" permission, and run the program.
9. Examples using the "chmod" command:
    - Example 1. To give yourself permission to execute a program, "file1", that you own:

      `chmod u+x file1`

    - Example 2. To give members of your group permission to read a file, "file2":

      `chmod g+r file2`

    - Example 3. To give read permission to everyone (all) for files with file extension ".pub", use chmod with "a", which indicates "all": "u", "g", and "o":

      `chmod a+r *.pub`

THIS PAGE INTENTIONALLY LEFT BLANK

II. INTRODUCTION TO MATLAB

A. TEXTBOOK

The textbook referenced in these notes is:

"***Getting Started with MATLAB5***" by Rudra Pratap
Oxford Press, 1999. ISBN 0-19-512947-4

B. HOW DOES MATLAB WORK?

- MATLAB commands can be manually entered one at a time on the command line.

- MATLAB *compiles* the command and then executes the command immediately.

    - A *compiler* converts the commands into machine language that the computer can understand. Execution of the commands is a separate step.
    - FORTRAN and other programming languages compile a set of commands (a program) together and all the program commands are executed sequentially.

- MATLAB stores the output of the command in a *workspace* in computer memory. The workspace can be accessed and modified by subsequent commands.

    - Programs in other languages make temporary use of computer memory and immediately after the program finishes the memory is cleared and released for other use.
    - The result of the computations or other program output must be written to the screen or written to a computer file (typically ASCII text). Commands to write the output to the screen/file must be included in the same program with the equations.

- MATLAB automatically assigns the *variable type* as *double precision, real or complex* numbers. This preallocates memory space. Double precision is extra bytes of memory which allows a greater number of significant digits to be stored.

- In other languages, the programmer must predefine the variable type as integer, real, double precision real, complex, double precision complex, or binary.
- In MATLAB, a real number or an integer is stores it as a double precision real number.
- If the value of a variable is changed to be complex, MATLAB automatically changes the variable type to be a double precision complex number.
- Examples of variable types:

  Complex number: 17.592345 - 157.662000***i***
  Real number: 17.592345
  Integer: 17 (not 17.0, which is a real number)
  Purely imaginary: 0.0 + 592.123456***i***

- MATLAB dynamically allocates variables as multi-dimensional arrays.

  - In other languages, the programmer must specify the exact size of array variables at the beginning of the program and the matrix size cannot be changed in the program.
  - MATLAB allows the user to change the size of the array variable at any time and add/delete rows and columns.
  - MATLAB provides built-in functions for array reshaping and other manipulations (text, section 3.1.3).

- MATLAB automatically displays the contents of variable to the screen, using a default format for number of significant digits displayed.

  - Using the semicolon ";" will suppress the screen display.
  - Complex numbers are displayed as in complex format, but if there is no imaginary part to the number, only the real part of the number is displayed.
  - Using the format command (text, p. 11), the user can change the automatic output format.

- MATLAB allows UNIX or PC operating system commands to be issued from the command line. Some commands to list files or change directory are built-

in commands (text, p. 14). For example, the following UNIX commands work in MATLAB: **`pwd`**, **`ls`**, **`ls -l`**, **`cd`**, **`cd ..`**

- Other operating system commands may be issued from the command line, but must be preceded by an exclamation point. Examples are:

```
!jot myprogram.m
!rm oldfile.txt
!cp /home/usr6/mr2020/plot_sounding.m myplotsnd.m
```

C. STARTING MATLAB

- Starting MATLAB from the UNIX menu creates a MATLAB command window (text, p. 9) and points to the user's *home directory* as the first directory in the MATLAB search path for program files. Use the `cd` command to change directory, as needed.

- From a UNIX shell (winterm), type ***`matlab`*** at the UNIX prompt. The command window points to the directory the user is in when starting MATLAB.

D. USING MATLAB AS A CALCULATOR

- At the prompt in the command window, a valid equation can be entered and computed. If no variable name is assigned, MATLAB uses the default variable name "ans" (meaning answer) to store the result in the workspace. Example:

```
>> 145/16  +  355.4*6

ans =
2.1415e+003
```

E. ASSIGNING VARIABLE NAMES

1. To store any value in memory for future use, a *variable name* is assigned to the number. The user refers to the variable name and the computer extracts the number from memory and uses it.

2. The form of an *assignment statement* is:

   VariableName = EXPRESSION

   where *expression* can be a constant, another variable or a formula (math expression). The variable name is *always* on the left side of the equal sign.

3. The equal sign in the assignment statement *does not* mean *equality*. The expression (right hand side) is evaluated and the result is *stored* in the memory location denoted by the variable name.

4. Variable names can be reused -- the new number replaces the current value in the memory location.

   - Example1.

     X = 5 … 5 is stored in memory allocated to variable X
     X = 125 … 125 replaces 5 as the value stored in memory for X

   - Example 2.

     N = 20
     N = N + 1 … expression N+1 is evaluated and the result (21) is stored in N, replacing the old value (20).

5. Variable names must start with a character (not a number). The maximum length is 31 characters. MATLAB is case sensitive; X and x are different variable names. The underscore can be used in a variable name. Examples:

   VALID: Sounding_OAK_21Jun99
   VALID: X12
   VALID: Pressure
   VALID: temperature_C
   INVALID: 21Jun99_sounding

6. Use only *one variable name* on the left side of the equal sign. Examples of valid and invalid assignment statements.

```
INVALID:   5 = N        … reason:  5 is not a valid variable name
VALID:     N = 5

INVALID:   A + 10.3 = LENGTH     … only one variable left of = sign
VALID:     LENGTH = A + 10.3
```

7. There can be only one equal sign in each assignment statement. To assign two variables to be equal to a third variable, do this in two assignment statements.

```
INVALID:   A = B = C       … only one equal sign in an assignment
VALID:     B = C
           A = B        (two statements)
```

F. WORKSPACE ENVIRONMENT

1. MATLAB stores workspace variables in active memory.
2. Variables are listed with the *who* or *whos* commands (see text, p. 14).
3. Workspace variables can be deleted with the *clear* command (p. 14).

G. SAVING THE WORKSPACE

1. MATLAB can save the workspace for future work sessions, using the *save* command (test, p. 12 and 69). The workspace is saved as a *binary* file that can *only* be read by MATLAB. The file extension of these binary files is "*.mat* ".
2. The user must define the file name for the binary ".mat" file.
3. The user may save all variables or specify certain variables to be saved to the ".mat" file.
4. The binary files can be quite big.

H. RELOADING THE WORKSPACE

1. The load command is used to load a binary ".mat" file into the active workspace (text, p. 12 and 69).

2. If the same variable name is used in the current workspace and in the ".mat" file to be loaded, the current value of the variable will be overwritten by the variable loaded in from the ".mat" file.

    - Example: x = 5 in the current workspace, but x = 25 in "file1.mat". After the command "load file1.mat", variable x contains the value 25.

    - If you do not know the variable names used in a ".mat" file, load the file into an empty workspace.

III. ORDER OF PRECEDENCE

1. Complicated equations are evaluated using ***rules*** called the ***order of precedence.***
2. A programmer needs to use sufficient numbers of parentheses to ***ensure*** the computer evaluates the equation in the order the programmer intends.
3. Parentheses are not always needed, but the programmer must know how the equation will be interpreted by MATLAB.
4. For example, how should MATLAB interpret the equation: `Z = X-4/Y`
    - Does the programmer mean: `Z = X-(4/Y)` , or `Z = (X-4)/Y`
    - Without any parentheses, MATLAB will compute: `Z = X-(4/Y)`
5. If the programmer intended `Z=(X-4)/Y`, but typed `Z=X-4/Y`, then the programmer will obtain an unanticipated result because MATLAB will compute precisely what is programmed. Hopefully, the programmer will properly test the program to notice this mistake. MATLAB will not generate an error message in this case. MATLAB was given a valid equation.
6. An error message will be generated if parentheses in the equation are not properly matched – equal number of left and right parentheses. MATLAB cannot understand the equation if parentheses are not matched.

7. Given an equation, the operations performed using the following rules:

    1. ***Parentheses are evaluated first***. For equations with multiple parentheses, the innermost parentheses evaluated first. Within each set of parentheses, expressions are evaluated using the three precedence levels listed below.

    2. There are ***three Precedence Levels***. Operations at the same level are preformed from Left to Right. Level 1 is the highest precedence.

        Level 1: **Power: ^**

        Level 2: **Multiplication: * , Division: /** (forward slash is "normal" division)

        Note: recommend students not use the other division operator \ (back slash).

        Level 3: **Addition: + , Subtraction: -**

    - **Order of Precedence Example:**

    $3\dfrac{\sqrt{5}-1}{\left(\sqrt{5}+1\right)^2}-1$ Text, p. 21, question 1.

    3* (sqrt(5) - 1) / ((sqrt(5) + 1) ^2) - 1 ... Correct MATLAB equation

    3* (sqrt(5) - 1) / (**(sqrt(5) + 1)** ^2) -1
    … innermost parentheses evaluated first

    3* **(sqrt(5) - 1)** / **(3.2361 ^2)** - 1 … outer parentheses evaluated second

    3* 1.2361 / 10.4723 - 1 ... multiplication evaluated third (Left operation)

    3.7083 / 10.4723 - 1 ... division evaluated fourth

    0.3541 - 1 ... subtraction evaluated last

    Answer: -0.6459

Note 1: To avoid worrying about Left to Right evaluation of multiplication and division (since they have equal precedence), add another set of parentheses to enclose the entire numerator:

(3* (sqrt(5) - 1) **)** / ((sqrt(5) + 1) ^2) - 1

(3* **(sqrt(5) - 1)**) / (**(sqrt(5) + 1)** ^2) - 1
... innermost parentheses evaluated first

(3* 1.2361) / **(3.2361 ^2)** - 1 ... outer parentheses evaluated second

3.7083 / 10.4723 - 1 …division evaluated third

0.3541 - 1 ... subtraction evaluated last

Note 2: Parentheses around the entire fraction would clearly identify that it is evaluated first before subtracting one:

((3* (sqrt(5) - 1)) / ((sqrt(5) + 1) ^2) **)** - 1

IV. MATRIX OPERATORS

A. OVERVIEW

Matrix operators follow the rules of Linear Algebra. The examples presented here are the typical uses of matrix operators in Meteorology and Oceanography. Consult MATLAB documentation for a complete list of matrix operators.

The following matrices have been defined. **A** and **B** are *square* matrices with dimensions 2 rows by 2 columns. **S** is a 1 row by 1 column matrix, called a **scalar**.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \qquad S = \begin{bmatrix} s_{11} \end{bmatrix}$$

B. MATRIX ADDITION AND SUBTRACTION

For matrices with equal dimensions (not restricted to square matrices), addition and subtraction of matrices is accomplished *term-by-term*.

$$A+B = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} \\ a_{21}+b_{21} & a_{22}+b_{22} \end{bmatrix} \qquad A-B = \begin{bmatrix} a_{11}-b_{11} & a_{12}-b_{12} \\ a_{21}-b_{21} & a_{22}-b_{22} \end{bmatrix}$$

Addition or subtraction of a *scalar* is also accomplished *term-by-term*. In the case of a scalar, the matrices do not need to be the same size.

$$A+S = \begin{bmatrix} a_{11}+s_{11} & a_{12}+s_{11} \\ a_{21}+s_{11} & a_{22}+s_{11} \end{bmatrix} \qquad A-S = \begin{bmatrix} a_{11}-s_{11} & a_{12}-s_{11} \\ a_{21}-s_{11} & a_{22}-s_{11} \end{bmatrix}$$

C. MATRIX MULTIPLICATION

Two matrices can be multiplied together only if the number of columns of the first matrix equals the number of rows of the second matrix.

- If X has dimensions 10x20, and Y has dimensions 20x7, then the multiplication of X*Y is defined and the result is a 10x7 matrix. In this case, Y*X is not allowed because the number of columns of Y (seven) does not match the number of rows of X (ten).

- For square matrices of equal dimensions, such as matrices A and B, both A*B and B*A are defined (but yield *different* results).

$$A*B = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

$$B*A = \begin{bmatrix} b_{11}a_{11} + b_{12}a_{21} & b_{11}a_{12} + b_{12}a_{22} \\ b_{21}a_{11} + b_{22}a_{21} & b_{21}a_{11} + b_{22}a_{21} \end{bmatrix}$$

Multiplication of a matrix of any size by a *scalar* is always defined, and S*A = A*S.

$$A*S = \begin{bmatrix} a_{11}s_{11} & a_{12}s_{11} \\ a_{21}s_{11} & a_{22}s_{11} \end{bmatrix}$$

D. MATRIX DIVISION

Matrix division, using either the forward (/) or backward (\) slash, is related to taking the *inverse* of a matrix while solving a set of simultaneous equations.

For A and B the same-sized square matrices:

- A / B equals $A*B^{-1}$ (note: B^{-1} means inverse of B)
- A \ B equals $A^{-1}*B$
- For simplicity, use the *forward* slash.

Division by a *scalar* matrix is defined *only* if the scalar is in the denominator:

- A / S is valid -- matrix A divided by a scalar number, s_{11}, is the same as multiplying A by the scalar quantity $(1/s_{11})$.
- S / A is *not* valid.
- S \ A is valid because it equals A / S.
- For simplicity, use the *forward* slash.

For more detailed information on matrix division, see the MATLAB on-line help topic *arithmetic operators*.

E. MATRIX EXPONENTIATION

Raising a matrix to a power is only defined for *square* matrices and uses the operator ^ (caret).

- A^2 means A^2, and is A*A.
- A^4 means A^4, and is A^3*A, or ((A*A) * A) *A.
- (A*D)^2 is only valid if A*D is a square matrix.
- Note: (A*B)^2 is not equal (A^2) * (B^2).

F. EXAMPLES USING MATRIX OPERATORS

Matrix Assignments

```
>> A = [1 2;3 4]
A =
     1     2
     3     4

>> B = [40 45; 50 55]
B =
    40    45
    50    55

>> S = 100
S =
   100
```

Addition and Subtraction

```
>> C = A+B
C =
    41    47
    53    59

>> D = A-B
D =
   -39   -43
   -47   -51

>> E = A+S
E =
   101   102
   103   104

>> F = A-S
F =
   -99   -98
   -97   -96
```

Matrix Multiplication

The matrix multiplication operator accomplishes in one command what FORTRAN, C, and many other languages need four commands to do. H = A*B is the equivalent of these four separate equations:

```
>> G(1,1) = A(1,1)*B(1,1) + A(1,2)*B(2,1);
>> G(1,2) = A(1,1)*B(1,2) + A(1,2)*B(2,2);
>> G(2,1) = A(2,1)*B(1,1) + A(2,2)*B(2,1);
>> G(2,2) = A(2,1)*B(1,2) + A(2,2)*B(2,2)

G =
   140   155
   320   355

>> H = A*B
H =
   140   155
   320   355
```

A*B is not equal to B*A:

```
>> K = B*A
K =
   175   260
   215   320
```

Multiplication by a scalar: A*S equals S*A.

```
>> L = A*S
L =
   100   200
   300   400
```

Matrix Division

For simplicity, use the forward slash. The backward slash has a different meaning. "inv" is the built-in function to take the inverse of a matrix.

```
>> M = A/B
M =
    0.9000   -0.7000
    0.7000   -0.5000

>> N = A*inv(B)
N =
    0.9000   -0.7000
    0.7000   -0.5000

>> P = A\B
P =
   -30   -35
    35    40
```

```
>> Q = inv(A)*B
Q =
  -30.0000  -35.0000
   35.0000   40.0000
```

Division by a scalar is valid, but dividing a scalar by a matrix is not valid:

```
>> R = A/S
R =
    0.0100    0.0200
    0.0300    0.0400
```

Matrix Exponentiation (Powers)

```
>> U = A^4
U =
   199   290
   435   634

>> V = A*A*A*A
V =
   199   290
   435   634
```

V. ARRAY OPERATORS

A. OVERVIEW

Array operations are unique to MATLAB and extremely powerful. Array operators do in *one* step what other languages accomplish in several lines of code. The key is that array operators perform *term-by-term* operations.

The following matrices have been defined. **A** and **B** are *square* matrices with dimensions 2 rows by 2 columns. **S** is a 1 row by 1 column matrix, called a **scalar**.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \qquad S = \begin{bmatrix} s_{11} \end{bmatrix}$$

B. ARRAY ADDITION AND SUBTRACTION

There is no difference between matrix and array addition and subtraction. For matrices with equal dimensions (not restricted to square matrices), addition and subtraction of matrices is accomplished *term-by-term*.

C. ARRAY MULTIPLICATION

Two matrices which are the same size can be multiplied together using the " **.*** " array operator. Matrices do not need to be square.

- Since the operator is applied term-by-term, A **.*** B and B **.*** A are equal.

$$A .* B = \begin{bmatrix} a_{11} * b_{11} & a_{12} * b_{12} \\ a_{21} * b_{21} & a_{22} * b_{22} \end{bmatrix}$$

- Multiplication of a matrix of any size by a *scalar* is always term-by-term, so there is no difference when using the " * " or " **.*** " operators. Technically, using " **.*** " with scalar multiplication is unnecessary, but the operator works.

$$A .* S = \begin{bmatrix} a_{11} s_{11} & a_{12} s_{11} \\ a_{21} s_{11} & a_{22} s_{11} \end{bmatrix}$$

D. ARRAY DIVISION

Array division, using either the forward (**.** /) or backward (**.**\ ) slash, is term-by-term division applied to same-sized matrices (not restricted to square matrices).

- A **.** / B and A **.**\ B (do not produce the same result)

$$A./B = \begin{bmatrix} a_{11}/b_{11} & a_{12}/b_{12} \\ a_{21}/b_{21} & a_{22}/b_{22} \end{bmatrix} \qquad A.\backslash B = \begin{bmatrix} b_{11}/a_{11} & b_{12}/a_{12} \\ b_{21}/a_{21} & b_{22}/a_{22} \end{bmatrix}$$

- For simplicity, use the *forward* slash (**.** /) operator.

Array division with a *scalar* matrix is defined when the scalar is in either the numerator or denominator.

- A **.** / S is the same as A / S. Technically, using " **.** / " when the scalar is in the denominator is unnecessary, but the operator works.

$$A./S = \begin{bmatrix} a_{11}/s_{11} & a_{12}/s_{12} \\ a_{21}/s_{21} & a_{22}/s_{22} \end{bmatrix}$$

- S **.** / A is valid. S / A is not defined, so the " **.** / " array operator must be used to compute term-by-term division.

$$S./A = \begin{bmatrix} s_{11}/a_{11} & s_{11}/a_{12} \\ s_{11}/a_{21} & s_{11}/a_{22} \end{bmatrix}$$

- The backward slash (**.**\ ) operator works with scalars, but for simplicity, use the *forward* slash (**.** /) operator.

E. ARRAY EXPONENTIATION

Raising a matrix to a power is defined for same-sized matrices and arrays and uses the operator " **.**^ ".

- A **.**^ B and B **.**^ A are both defined, but produce different results.

$$A.\hat{}\,B = \begin{bmatrix} a_{11}^{\,b_{11}} & a_{12}^{\,b_{12}} \\ a_{21}^{\,b_{21}} & a_{22}^{\,b_{22}} \end{bmatrix} \qquad B.\hat{}\,A = \begin{bmatrix} b_{11}^{\,a_{11}} & b_{12}^{\,a_{12}} \\ b_{21}^{\,a_{21}} & b_{22}^{\,a_{22}} \end{bmatrix}$$

- A matrix raised to a scalar, A **.^** S, requires the use of the " **.** ^ " operator. It is *not* the same as A ^ S, which is a matrix operation.

$$A .\hat{} S = \begin{bmatrix} a_{11}^{s_{11}} & a_{12}^{s_{11}} \\ a_{21}^{s_{11}} & a_{22}^{s_{11}} \end{bmatrix}$$

- A scalar raised to a matrix, S .^ A, requires the use of the " **.** ^ " operator.

$$S .\hat{} A = \begin{bmatrix} s_{11}^{a_{11}} & s_{11}^{a_{12}} \\ s_{11}^{a_{21}} & s_{11}^{a_{22}} \end{bmatrix}$$

F. NONCONJUGATED TRANSPOSE

The nonconjugated transpose operator, " **.'** ", applies to matrices of complex numbers. Given C, a matrix of complex numbers:

$$C = \begin{bmatrix} c_{11} + d_{11}i & c_{12} + d_{12}i \\ c_{21} + d_{21}i & c_{22} + d_{22}i \end{bmatrix}$$

- The transpose of C, denoted C**'**, reverses the rows and columns and takes the complex conjugate of each term.

$$C' = \begin{bmatrix} c_{11} - d_{11}i & c_{21} - d_{21}i \\ c_{12} - d_{12}i & c_{22} - d_{22}i \end{bmatrix}$$

- The non-conjugated transpose of C, denoted C **.'** , reverses the rows and columns but does not compute the complex conjugate of each term.

$$C.' = \begin{bmatrix} c_{11} + d_{11}i & c_{21} + d_{21}i \\ c_{12} + d_{12}i & c_{22} + d_{22}i \end{bmatrix}$$

G. EXAMPLES USING ARRAY OPERATORS

Matrix Assignments

```
>> A = [1 2;3 4]
A =
     1     2
     3     4

>> B = [40 45; 50 55]
B =
    40    45
          50    55

>> S = 100
S =
   100

>> A .* B       % this is the same as B .* A

    40    90
   150   220

>> A * S    % matrix or array times a scalar does not need " .* "

   100   200
   300   400

>> A ./ B       % forward slash -- easiest to remember

   0.0250    0.0444
   0.0600    0.0727

>>  A .\ B         % backward slash

   40.0000   22.5000
   16.6667   13.7500

>> A(:,1) ./ B(:,2)  % dividing same-sized column vectors

   0.0222
   0.0545

>> A / S        % division by a scalar does not need " ./ "

   0.0100    0.0200
   0.0300    0.0400

>> S ./ A   % scalar divided by a matrix or array requires " ./ "

   100.0000   50.0000
    33.3333   25.0000
```

```
>> A .^ B           % not the same as B .^ A

       1       3.5184e+013
   7.179e+023  1.2981e+033

>> B .^ A

        40        2025
    125000     9150625

>> S .^ A            % requires " .^ " operator

       100       10000
   1000000   100000000

>> A .^ S            % requires " .^ " operator

  1.0000e+000  1.2677e+030
  5.1538e+047  1.6069e+060

>> C = [10+4i 3-2i; 20-6i 0+2i]  % define complex number matrix
C =
  10.0000 + 4.0000i   3.0000 - 2.0000i
  20.0000 - 6.0000i        0 + 2.0000i

>> C'             % " ' " is the complex conjugate transpose

  10.0000 - 4.0000i  20.0000 + 6.0000i
   3.0000 + 2.0000i        0 - 2.0000i

>> C.'            % " .' " is the non-complex conjugate transpose

  10.0000 + 4.0000i  20.0000 - 6.0000i
   3.0000 - 2.0000i        0 + 2.0000i
```

VI. SIMPLE DATA INPUT AND OUTPUT

A. METHODS TO INPUT DATA INTO MATLAB

1. Enter data values manually from the command line.

1. Use the "input" built-in function.

2. Use the "load" built-in function:

    - Load a binary ".mat" file.
    - Load an ASCII text file with data in columns.
        i. File without text header lines (i.e. only has data)
        ii. File with text header lines with "%" in the first column (MATLAB will skip lines starting with the "%" symbol).

3. Read data from an ASCII file with "fscanf" or other built-in functions.

B. "input" FUNCTION

1. The "input" built-in function prompts the user to enter data and stores the input values in the specified variable name.

    - The function prints a specified character string to the screen, and waits for the user to enter the data.

    - Multiple data values may be entered at one time by defining a vector (data in square brackets).

    - If the user strikes the "Return/Enter" key without an input value, MATLAB will automatically assign empty matrix to the variable.

    - Unless specified to be a character string, MATLAB will interpret the input value as a number.

```
x = input('prompt')
```

2. There are two ways to enter a character string using the "input" function.

    - Specify the input be interpreted as a character string using the **'s'** option.

```
x = input('prompt','s')
```

- Do not use the **'s'** option, but enclose the character string in single quotes.

Example 1:

A. Input one numerical value:

```
>> T = input('Enter the temperature (C): ')

Enter the temperature (C): 18.3

T =
   18.3000
```

B. Input multiple numerical values:

```
>> TD = input('Enter the dew point (C): ')

Enter the dew point (C): [14.5 17.2 13.5]

TD =
   14.5000   17.2000   13.5000
```

Example 2:

Input *one* character string *without* using single quotes:

```
>> filenm = input('Enter the file name: ', 's')
Enter the file name: sounding_10Jan2002.txt

filenm =
sounding_10Jan2002.txt
```

Example 3:

Input *one* character string using single quotes:

```
>> filenm = input('Enter the file name: ')

Enter the file name: 'sounding_10Jan2002.txt'

filenm =
sounding_10Jan2002.txt
```

Example 4:

Input *multiple* character string using single quotes:

```
>> filenm = input('Enter the file name: ')

Enter the file name: ['snd1.txt';'snd2.txt']

filenm =
snd1.txt
snd2.txt
```

C. DISPLAYING VALUES TO THE SCREEN

1. MATLAB automatically displays the contents of variables to the screen, unless the output is suppressed with a semicolon (text, p. 10).

2. The "**format**" command is used to specify the format in which numbers are automatically displayed on the screen (test, p. 11).

3. To display the values stored in a variable in memory, type the name of the variable without a semicolon. For a matrix or array, all values are displayed to the screen.

4. Text strings and matrices can be displayed to the screen using the "**disp**" built-in function, with the format as specified with the "format" command.

5. The "**sprintf**" command provides the user with specific control over the format of display numbers and text displayed to the screen. This command is covered later in these notes.

D. "disp" FUNCTION

1. The "**disp**" function displays a matrix, without printing the matrix name. If the matrix contains a text string, the string is displayed.

2. Text strings may be displayed using the "disp" function, without defining the string in a matrix.

3. The "disp" function is useful in programs (M-files) to display messages to the screen during program execution.

4. Typical usage: **`disp(X)`**

Example 1: X is a 3x2 matrix.

```
>> X=[10.5, 100.2; -9.25, 0; 22.3, -75.66]

>> disp(X)

10.5000  100.2000
-9.2500         0
22.3000  -75.6600
```

Example 2: X is a row vector.

```
>> X=[10.5, 100.2 -9.25, 0]

>> disp(X)

10.5000  100.2000   -9.2500         0
```

Example 3: X is a column vector.

```
>> X=[10.5; -9.25; 22.3]

>> disp(X)

10.5000
-9.2500
22.3000
```

Example 4: X is a text string.

```
>> X=['This is a text string']

>> disp(X)

This is a text string
```

Example 5: Display a text string without defining a matrix.

```
>> disp('This is a text string')

This is a text string
```

Example 6: Messages displayed during execution of a program (M-file).

```
disp('loading data')
    more lines of code
disp('calculating potential temperature')
    more lines of code
disp('ending the program')
```

Example 7: Display a matrix of data with column labels. "temp" and "dewpt" are column vectors with the same number of elements.

```
disp('Temperature   Dew Pt')

disp([temp  dewpt])
```

```
  Temperature   Dew Pt
      16.0000   7.0000
      14.0000   3.0000
      12.4000   3.4000
      11.6000   3.6000
      11.1600   3.1600
       9.1900   1.1900
       8.0000        0
       6.9900  -0.4200
       4.4300  -1.4800
       2.2000  -2.4000
```

VII. SEARCH PATH

A. WHAT IS A "SEARCH PATH"?

1. MATLAB has a **search path**, which is the list of locations to search for M-files. If you enter a command, such as **`>>fox`**, MATLAB will:

    a. Look for *`fox`* as a variable.
    b. Check *`fox`* as a built-in function.
    c. Look in the current directory for *`fox.m`*.
    d. Search the directories specified by **path** for *`fox.m`*.

2. The "path" built-in function prints out the current setting of MATLAB's search path. **`>> path`**

3. The search path list originates from the environment variable "*matlabpath*", which has been defined to point to one user local directory, and the subdirectories containing MATLAB built-in functions and toolboxes.

4. Directories in the path are searched in order listed by "*matlabpath*".

5. The first directory listed in the search path is the user's home directory:

    - On IDEA and Graphics Lab UNIX workstations:

      `/h/mrhome1/username`

    - On a campus network PC: **`H:\username`**

    - On a personal PC, with MATLAB installed on the hard drive, the start directory is defined in the start-up file. Typically this location is:

      `C:\matlabr12\work`

6. Users may need to add directories to the search path to access user-written or course-provided programs.

7. On a personal PC, with MATLAB installed on the hard drive, the user may permanently add paths, which will be automatically available the next time MATLAB is started.

8. The default settings for student accounts using the NPS network MATLAB 6.0 software, both in UNIX and Windows 2000, does not allow paths added during one session can be saved and automatically available the next time the

user starts MATLAB. Student accounts can be modified to allow this to occur. See the course instructor for details.

B. ADD OR REMOVE A PATH

1. The "**addpath**" built-in function provides a temporary change to the path and available until the user ends the MATLAB session. This function may be invoked at the command line or in a program (M-file).

```
addpath  H:\username\matlabprograms\
```

2. Unless otherwise specified, the path is added as the ***last*** directory searched. To make the added path the ***first*** directory searched, use the option **`-begin`**.

```
addpath  H:\username\matlabprograms\  -begin
```

3. If a particular user program calls programs located in another directory, using the "addpath" function in the user program will add the other directory to the current path. The examples above will work in a program.

4. Typically, a user will only want to remove local directories from the path, not directories containing built-in functions. Use the "**rmpath**" built-in function. Since "addpath" is only a temporary change, students may never need to remove a path.

```
rmpath  H:\username\matlabprograms\
```

5. The "File" menu in the upper left corner of the MATLAB 6.0 command window contains a menu called "Set Path". This menu allows the user to add and remove paths, as well as change the directory order of the search path.

    - The option to ***save*** the new path settings works if MATLAB resides on the PC hard drive. The save option will work on UNIX or campus PC network versions on MATLAB 6.0 only if student accounts are modified. See the course instructor for details.

VIII. M-FILES

A. WHAT IS AN "M-FILE"?

- M-files are ordinary ASCII text files, which contain a set of MATLAB commands to be executed in the order listed. An M-file is a MATLAB program.
- The *file extension* of all M-files **must** be "**.m**". Example: `lab4.m`
- MATLAB will only recognize file names ending with ".m" as a program to execute (e.g., if MATLAB commands are listed in "`lab4.txt`", MATLAB will not execute the commands in this file).
- Use a text editor to create an M-file.
- There are two types of M-files:
  - script (regular) files
  - function files

B. SCRIPT M-FILES

- Refer to the textbook, section 4.1, pp. 77-80.
- Execute a **script** M-file by typing the name of the file (with the ".m") on the command line. This is the equivalent of typing each command in the file, one-at-a-time, at the command prompt.
- To execute a script M-file called `lab4.m`, type:

  `>>lab4` (not `lab4.m`)

- The script file uses variables already defined in the workspace. Additional variables defined in the script are added to the current workspace and are available for use after the script finishes execution. For example:
  - Workspace variables: A, B, C
  - Script M-file, `lab4.m`, defines variables X, Y and Z.
  - After `lab4.m` finishes execution, the workspace variables are A, B, C, X, Y, Z.
- Script M-files may include lines of code which execute other script or function M-files.

C. FUNCTION M-FILES

- The first two lines of a function file have a mandatory format, which distinguishes it from a script M-file.
- Functions are programs that can be executed many times. Storing the code in a file prevents a programmer from repeatedly adding the same lines of code to a script file.
- Functions are *executed* differently than a script M-file.
- A function uses variables already defined in the workspace.
    - Only variables identified to be output from the function are stored in the current workspace.
    - Additional variables defined in the function, but not designated for output, are ***not*** added to the current workspace.
    - All other variables in the function workspace are deleted after the function executes.
- Functions will be covered in great detail later in the course notes.

D. RULES APPLYING TO BOTH SCRIPT AND FUNCTION M-FILES

- The names of an M-file must begin with a letter, and can be no longer than 19 characters.
- Do not use a period anywhere in the name except ".m".
- Avoid script names that clash with built-in function names, (sin, cos, etc.) or predefined variable names, such as pi.
- Never name the script or function file the same name as the variable it computes (text, p. 79).
- Programmer comments can be added to MATLAB code. Use the percent sign (%) preceding any comments in the code.
    - Typically % is in column 1 of a comment line.
    - A short comment can be added after the command. For example:

```
mix_ratio=(0.622*e_mb./(Pres -e_mb);   % mix_ratio in kg/kg
```

- The M-file must be located in directory listed in the MATLAB path. Otherwise, MATLAB will not be able to find the file.

E. HOW TO CREATE AN M-FILE

- M-files can be created and edited with any ASCII text editor. The file is saved as text with ".m" as the file extension.
    - In the IDEA and Graphics Labs, use "jot" or "nedit" text editors.
    - In Windows, use the "Notepad" text editor.
- MATLAB version 6.0 has a built-in file editor.
    - Use the "open folder" icon to open an existing M-file.
    - Use the "blank page" icon to open a blank (new) M-file.
    - MATLAB's text editor uses color to highlight comment lines, text strings, certain programming commands, such as "if", "else", and "end". This is helpful for a programmer.
- There is no difference between files created with a built-in or external text editor. Use the editor that is easiest for you.
- Since the MATLAB 6.0 is part of the campus network, sometimes opening, closing, and saving files may be slow if the network is slow.
- Save your work every few minutes, in case of a computer problem.
- At times, a network crash will cause you to lose the file you are currently editing. If it is an extremely important file, save it occasionally with a different file name. At worst, you may lose the current file, but an older version, with a different file name, still exists because it was not open at the time of the crash.

F. WHY USE A SCRIPT M-FILE?

- A script M-file executes faster than if each command was entered one-by-one at the command line.
- An M-file provides a record of the commands used to produce a result.
- M-files can be edited to correct mistakes or modify for additional use.
- M-files can be shared with other users.

G. CREATING AND TESTING A SCRIPT M-FILE

- The copy and paste options allow commands to be copied from the command window to and from the M-file. This makes creating and testing easy.

- This is an example of how to create and test a script M-file:
    - Clear your workspace.
    - Type a few commands one-by-one in the command window.
        - Debug the commands, as necessary.
    - Copy each command to a blank text editor file.
        - Make sure the MATLAB prompt ">>" is removed from the M-file. Copy only the command, not the workspace result of the command.
    - Save the file as an M-file.
    - Clear your workspace and execute the M-file.
        - If the results are not correct, make corrections to the M-file. Then, clear your workspace and re-execute the M-file.
        - Don't continue building your M-file until the first set of commands works.
    - Create and test more commands in the command window, then copy them to the M-file. Test and debug the M-file.
    - Continue this process until the M-file is completed. Then, clear the workspace and test it one last time.
    - Document the purpose of the program, variable names: meaning and units (knots, m/s, etc.), and program logic.
- Selected portions of a script M-file may be tested by copying the files from the M-file and pasting them, which executes the block of code, in the command window.

H. PROGRAMMING TIP

While a script M-file will can access whatever variables exist in the workspace when it is executed, it is smart programming form to include variable assignment statements, load commands, etc., in the M-file itself.

- This will allow the script M-file to execute independently in an empty workspace.
- Provides a record of which ".mat" files, or data sets the script is designed to use.
- If the workspace must be externally generated (workspace already exist before the script is executed):
    - The programmer should consider whether the script M-file should be restructured to become a function M-file.

- Extensive documentation about the workspace requirements must be included in the script M-file.

I. DOCUMENTATION REQUIREMENTS FOR SCRIPT M-FILES

- All script M-files for this course must have the following documentation information at the top of each file:
  - Comment lines start with the percent sign (%) in column 1
  - Line 1: name of the script M-file (e.g., file = lab4.m)
  - Line 2: student name and date
  - Line 3: course number and lab number (if appropriate)
  - Purpose of the program.
  - List of variable names, description and units (e.g., knots, m/s, etc.)
  - List of functions called in the script (if none, state it).
- Additional comment lines may be needed throughout the program to explain the logic.
- If an equation in the script is from a textbook or journal article, use comments lines to cite the title, equation number and page number.
- All comment lines in the header block at the top of an M-file must start with a % sign, even otherwise blank lines.
  - The help function will print out the header lines of an M-file until the first line of the program which does not start with %.
  - `>>help lab5.m` will print the header block of the example below.
- Blank lines do not need to start with a % sign. Blank lines may be added to improve readability of the program.

- Example:

```
% filename = lab5.m
% Mary Jordan, 9 Oct 96
% MR2020, Lab #5
%
% Purpose: Calculate the mean value of X, store it in Y.
%
% Variables:
%           X = data vector
%           Y = mean of X
%           m = length of X
%
% Functions used:
%           length.m (built-in function)
%           sum.m (built-in function)
%

X=[10,14,20,-1,25.3,-3,5,6.6];
m = length(X);
Y = sum(X)/m;
```

IX. RELATIONAL AND LOGICAL OPERATORS USING SCALAR VARIABLES

A. INTRODUCTION

- The relational operators and logical operators are used differently if the comparison is between ***scalars*** or ***equal-sized arrays***.
    - The explanation *in this section* pertains to ***scalar*** variables.
    - The examples in the textbook, sections 3.2.2 and 3.2.3, apply relational and logical operators to *arrays*. This topic is covered later in this course.
- Relational operators and logical operators are used in expressions which are part of "while" loops and "if-elseif-else" branching.
    - If the expression is true, then specified portions of the program are executed. Otherwise, those portions of code are ignored.

B. DEFINITIONS

- ***Logical expressions*** have a value of ***true*** or ***false***.
    - ***true*** is assigned a numerical value of **1**.
    - ***false*** is assigned a numerical value of **0**.
- Logical expressions are defined using relational and logical operators and scalar variables.
- At first glance, logical expressions look like equations. While equations assign values to a variable, logical expressions are *comparisons* between variables.
- The ***relational operators*** are *comparisons* between two scalar variables or a number and one scalar variable.
    - The result of the comparison is *true* or *false* (1 and 0, respectively).

- The ***relational operators*** are:

<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
= =	Equal (no space between = =)
~=	Not equal (tilde)

- The ***logical operators*** are:

&	And
|	Or
~	Not

- The logical operators "And" and "Or" compare two expressions, each expression being either *true* or *false*, and the result is a final value of either *true* or *false*.

    - For the "And" operator, ***both*** expressions must be true for the comparison to be true.

true & true	result is ***true***
true & false	result is ***false***
true & false	result is ***false***

    - For the "Or" operator, ***only one*** expression must be true for the comparison to be true.

true | true	result is ***true***
true | false	result is ***true***
false | true	result is ***true***
false | false	result is ***false***

- The logical operator "Not" acts on one expression, its value being either *true* or *false*, and the result is to reverse the value of the expression from *true* to *false*, and *false* to *true*.

~ true	result is ***false***
~ false	result is ***true***

Example 1: Relational operators: Comparing one variable.

- x < 5 means "is x less than 5?". The answer is true or false (1 or 0).

Assume x = 14.
Expression: x < 5 … is evaluated as false (zero).
Assigned to a variable: A = x < 5 … zero (false) is stored in A.

Example 2: Relational operators: Compound expressions with AND

- Evaluate the equation: $-10 \le x < 22$
- MATLAB expression: **`(-10 <= x) & (x < 22)`**
- This is a compound expression, and each part is evaluated separately.

Assume x = 14.

```
-10 <= x                ... -10 <= 14   is true
  x < 22                ... 14 < 22     is true
-10 <= x  &  x < 22     true & true     … result is true
```

Expression: `(-10<=x) & (x<22)` … is evaluated as true (1).

Assigned to a variable: **`B=(-10<=x)&(x<22)`** … 1 (true) is stored in B.

Example 3: Relational operators: Compound expressions with OR

- Evaluate the equation: $-10 \le x \ \ OR \ \ y < -50$
- MATLAB expression: **`(-10 <= x) | (y < -50)`**
- This is a compound expression, and each part is evaluated separately.

Assume x = 14 and y = -25.

```
-10 <= x            …     -10 <= 14   is true
 y < -50            …     -25 < -50   is false
-10 <= x  | y < -50       …. true | false  … result is true
```

Expression: `(-10<=x) | (y<-50)` … is evaluated as true (1).

Assigned to a variable: **`C=(-10<=x)|(y<-50)`** … 1 (true) is stored in C.

Example 4: Comparisons which include variable expressions

- Evaluate the equation: $z < (10x + \frac{y^2}{3})$
- MATLAB expression: **`z < (10*x + (y^2)/3)`**
- Evaluate the variable expression (equation) and compare the result to the value of z.

Assume x = 14, y = -25, z = 500.

`(10*x + (y^2)/3)` … 348.3

500 < 348.3 … is false

Expression: `(10*x + (y^2)/3)` … is evaluated as false (0).

Assigned to a variable: **`D= z<(10*x+(y^2)/3)`**… 0 (false) is stored in D.

X. LOOPS, BRANCHES AND CONTROL-FLOW

A. INTRODUCTION

- There are basic program logic control-flow and branching statements in MATLAB, which are similar to logic in other languages. MATLAB has slightly different construction and syntax, but the logic flow is the same.
- A "for" loop is a set of code executed once using each value assigned to the "for" loop variable.
- A "while" loop executes while a specified expression is logically true. When the expression is false, the commands in the loop are not executed.
- An "if-elseif-else" branching is executed only if specified expressions are true.
- The "while" loop and "if-elseif-else" branching evaluates expressions using scalar variables which are compared using relational and logical operators, described in the previous section.

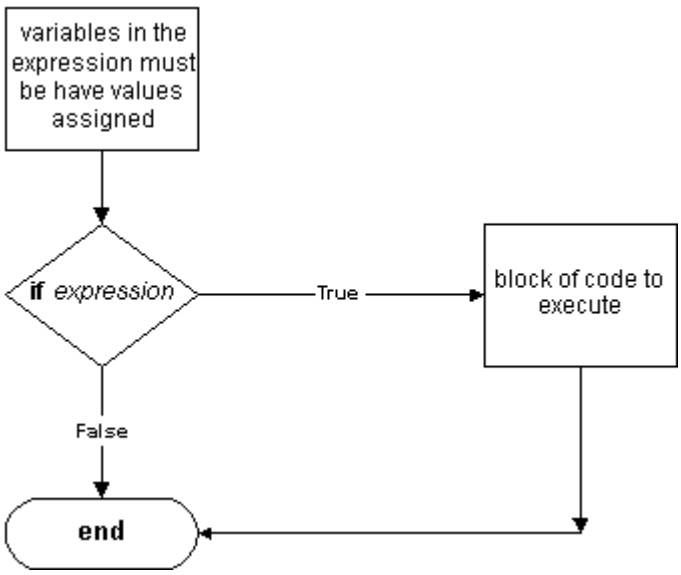
B. " if " STATEMENTS

1. Basic "if" Statement

Format of code:

```
if expression
   statement block
end
```

Figure 1. Flow chart for simple "if" statement.



Execution:

- The variables in the expression to be tested must have values assigned prior to entering the "if" statement.
- The block of code will only execute if the expression is true. If the expression is false, the code is ignored.
- Values assigned to the variables used in the expression may be changed in the block of code inside the "if" statement.

Example:

```
x = 100;
y = -10;

if  x < 150
   z =  x - y^2;
   A = x + 12.5*y + z^3;
end
```

1. "if else" Statement

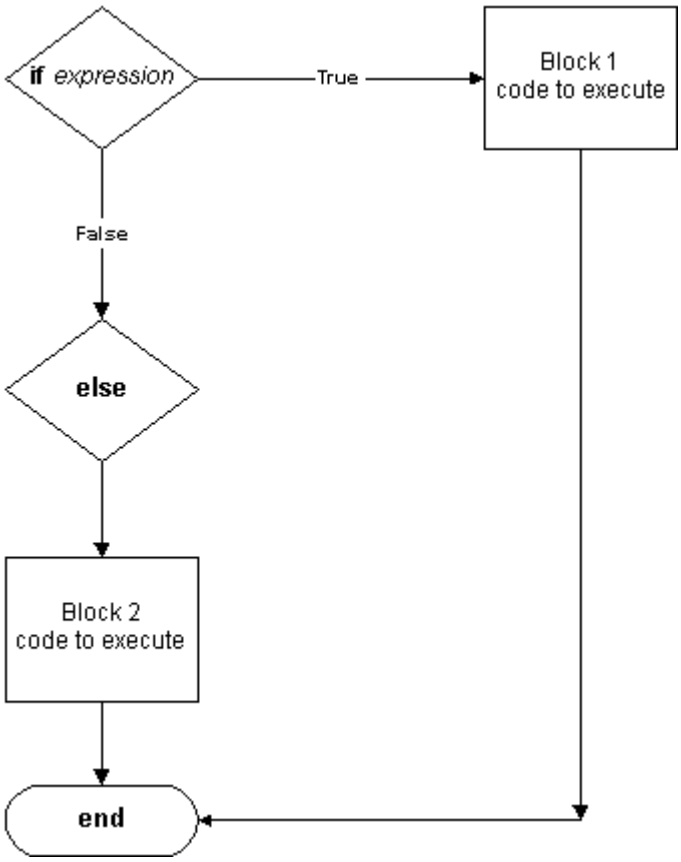
Format of code:

```
if expression
   statement block 1
else
   statement block 2
end
```

Execution:

- The variables in the expressions to be tested must have values assigned prior to entering the "if" statement.
- The block #1 of code will only execute if the expression is true.
- If the expression is false, the block #1 code is ignored and the code in block #2 is executed.
- Values assigned to the variables used in the expression may be changed in the block of code inside the "if" statement.

Figure 2. Flow chart for "if else" statement.



Example:

```
x = 100;
y = -10;

if  x < 150
   z = x - y^2;
   A = x + 12.5*y + z^3;
else
   z = 2*x + y;
   A = (x/12.5) - 3*y + z/2;
end
```

2. "if elseif else" Statement

Format of code:

```
if expression1
   statement block 1
elseif expression2
   statement block 2
else
   statement block 3
end
```

Execution:

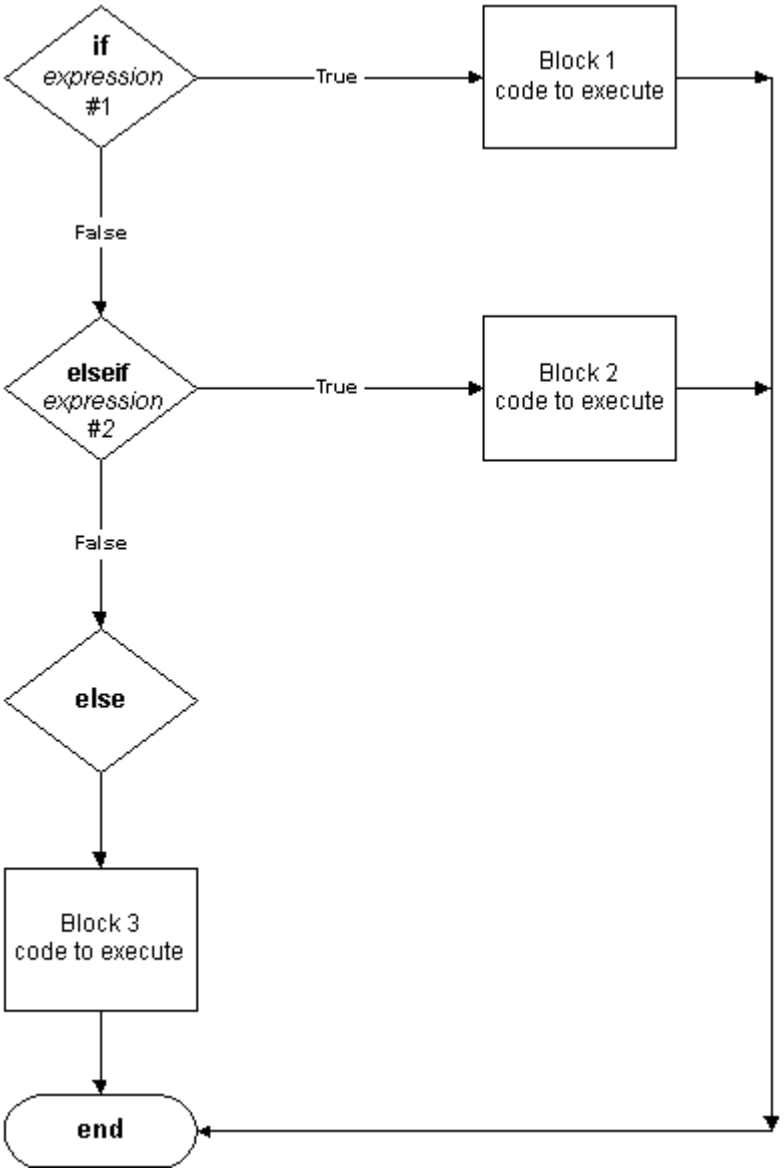
- The variables in the expressions to be tested must have values assigned prior to entering the "if" statement.
- The block #1 of code will only execute if the expression #1 is true.
- If the expression #1 is false, the block #1 code is ignored and expression #2 is evaluated.
- The code in block #2 is executed only if expression #2 is true.
- If the expression #2 is false, the block #2 code is ignored and code in block #3 is executed.
- Only *one* block of code is executed and transfer of control is to the **end** statement. All other lines of code inside the "if" statement are ignored.
  - Even if subsequent expressions are also true, MATLAB never allows those statements to be evaluated.
    - If expressions 1 and 2 are both true, block #1 is executed and expression #2 is not even checked.
  - Values assigned to the variables used in the expression may be changed in the block of code inside the "if" statement.

Example:

```
x = 100;
y = -10;

if  x < 150
   z = x - y^2;
   A = x + 12.5*y + z^3;
elseif x > 150 & x <= 200
   z = 2*x + y;
   A = (x/12.5) - 3*y + z/2;
else
   z = 65 - x/2;
   A = x - y + z^2;
end
```

Figure 3. Flow chart for "if elseif else" statement.



3. "if elseif " (no else) Statement

Format of code:

```
if expression1
   statement block 1
elseif expression2
   statement block 2
elseif expression3
   statement block 3
elseif expression4
   statement block 4
end
```

Execution:

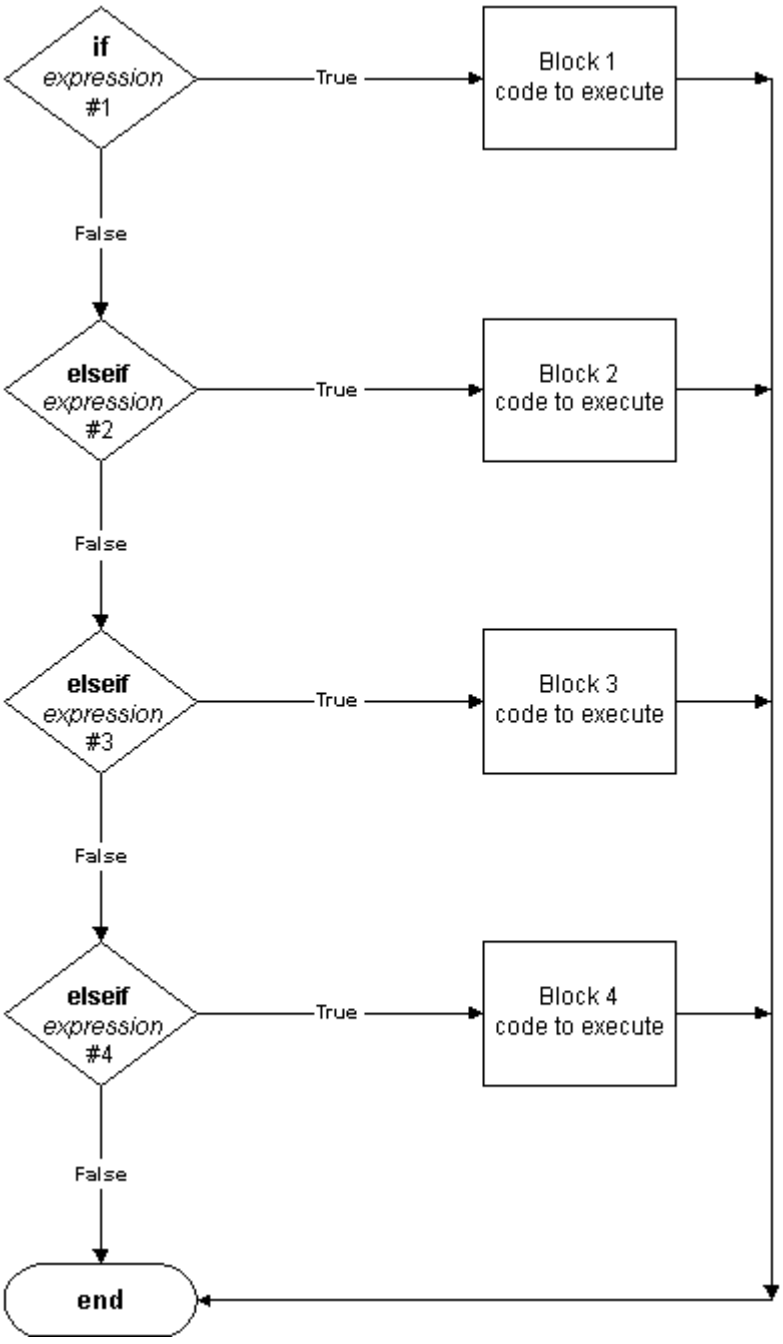
- The variables in the expressions to be tested must have values assigned prior to entering the "if" statement.
- There is no **else** statement, so no block of code will be automatically executed if all expressions are false.
- The block #1 of code will only execute if the expression #1 is true.
- If the expression #1 is false, the block #1 code is ignored and expression #2 is evaluated.
- The code in block #2 is executed only if expression #2 is true.
- If the expression #2 is false, the block #2 code is ignored and expression #3 is evaluated, and so on.
- Only *one* block of code is executed and transfer of control is to the **end** statement. All other lines of code inside the "if" statement are ignored.
- Values assigned to the variables used in the expression may be changed in the block of code inside the "if" statement.

Example:

```
x = 100;
y = -10;

if  x < 150
   z = x - y^2;
   A = x + 12.5*y + z^3;
elseif x > 150 & x <= 200
   z = 2*x + y;
   A = (x/12.5) - 3*y + z/2;
elseif y > 0
   z = sqrt(y);
   A = x - 3*y + z;
elseif y <= 0
   z = sqrt(-y);
   A = x - 3*y + z;
end
```

Figure 4. Flow chart for "if elseif " (no else) statement.



C. “ for ” LOOPS

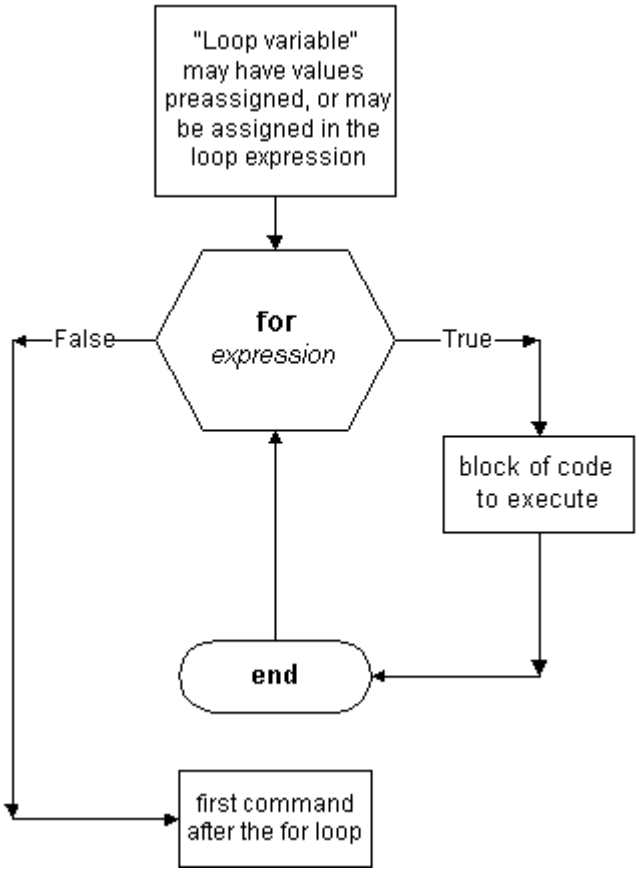
1. Basic "for" loop

Format of code:

```
for variable = expression
     statements to be executed, also called the
     "body of the loop"
end
```

2. Flow chart

Figure 5. Flow chart of a basic “for” loop.



3. “for” Loop Rules

- "for" loops end with the "end" statement (both are lower case).
- The values the *loop variable* are controlled by the expression.

- The first time through the loop, the variable is assigned the first value in the expression. For the second time through the loop, MATLAB *automatically* assigns to the variable the second value in the expression. This continues for each value in the expression. The loop is automatically terminated after the body of the loop has been executed with the loop variable assigned to each value in the expression.

- The body of the loop is executed many times, but the loop is only executed *once* for each value in the expression.

- The user *cannot change* the value of the loop variable in any statement in the body of the loop. Once the loop starts executing, MATLAB controls values assigned to the loop variable and will not permit the user to change it.

- After the loop is finished executing, the loop variable still exists in memory and its value is the last value used inside the loop.

- If the name of the loop variable was already used in the program prior to execution of the loop, old values of the variable are erased and the values of the variable are controlled by the loop.

- Any name can be used for a loop variable.

- Many programmers use i, j or k as their loop variable names. These loop variable names should not be used if the program uses complex numbers (for complex numbers, i and j are predefined as the square root of -1).

- The *expression* in a for loop is an *array* of values (real or complex). The number of times the loop executes equals the number of values in the expression array.

- The values in the expression array do not have to be integers.

- Typical use of a loop is to count the number of times a set of statements is executed.

- Loops can be nested.

Example 1: Counting with a Loop.

- This loop is used to execute the body of the loop a specified number of times.
- Loop variable, i, is defined with sequential numbers.

```
for i = 1:3         % execute three times
   x = i^2
end

x =
     1
x =
     4
x =
     9
```

Example 2: Loop variable with nonsequential numbers.

- This loop is used to execute the body of the loop a specified number of times.
- Loop variable, i, is defined with nonsequential numbers.

```
for i = 5:10:35           % expression executes four times
    i
end

i =
     5
i =
    15
i =
    25
i =
    35
```

Example 3: Nested Loops.

- The outer loop variable is **i** and the inner loop variable is **j**.
- The inner loop is completed for each time the outer loop is incremented.

```
for i = 1:3                 % Outer loop
   i
   for j = 10:10:30         % Inner loop
      j
   end                      % end for inner loop
end                         % end for outer loop
```

```
i =
     1
j =
    10
j =
    20
j =
    30
i =
     2
j =
    10
j =
    20
j =
    30
i =
     3
j =
    10
j =
    20
j =
    30
```

Example 4: Using a loop to populate an array.

- One element of the array is computed and assigned during each execution of the loop.
- The loop variable is defined with sequential numbers.

```
for j = 1:10
    Y(j) = j ^ 2;
end

Y
Y =
     1     4     9    16    25    36    49    64    81   100
```

Example 5: Using nested loops to populate a matrix.

- The inner and outer loop variables, i and j, are used as the row and column indices of the matrix. A 5x3 matrix will be populated.
- One element of the matrix is computed and assigned during each execution of the loop.
- The loop variables are defined with sequential numbers.

```
for i = 1:5
   for j = 1:3
      X(i,j) = i^2 + j^2;
   end
end

X
X =
     2     5    10
     5     8    13
    10    13    18
    17    20    25
    26    29    34
```

Example 6: Use an array in the expression for the loop variable.

- The loop variable, **i**, is defined with nonsequential numbers in array, **n**.

```
n = [1 19 31 6];
for i = n
   i
end

i =
     1
i =
    19
i =
    31
i =
     6
```

Example 7: Use nonsequential indices to populate an array.

- The loop variable, **i**, is defined with nonsequential numbers in array, **n**.
- The array elements not defined in the loop are automatically assigned the value zero.
- This method to populate an array is not used often.

```
n = [1 19 31 6];
for i = n
   Z(i) = i;
end
```

```
Z
Z =
  Columns 1 through 12
     1     0     0     0     0     6     0     0     0     0     0     0
  Columns 13 through 24
     0     0     0     0     0     0    19     0     0     0     0     0
  Columns 25 through 31
     0     0     0     0     0     0    31
```

Example 8: Incrementing a counter without using the loop variable.

- The variable **count** will be incremented and as a sequential counter.
- The value of count is initialized as zero prior to the execution of the loop.
- The counter is incremented in the body of the loop.
- Any variable name can be used as a counter.

```
count = 0;

for i = 100: -3: 85
   count = count + 1
end

count =
     1
count =
     2
count =
     3
count =
     4
count =
     5
count =
     6
```

Example 9: Use a counter as an array index.

- The variable **count** will be incremented and used as the index of the array being populated.
- The loop variable, which has nonsequential values, is used in an equation to assign each array element value.

```
count = 0;                          % counter initialized outside loop

for i = 100: -3: 85
   count = count + 1;               % counter incremented
   X(count) = i * sin(pi/4);        % counter used as array index
end

X
X =
   70.7107   68.5894   66.4680   64.3467   62.2254   60.1041
```

D. " while " LOOPS

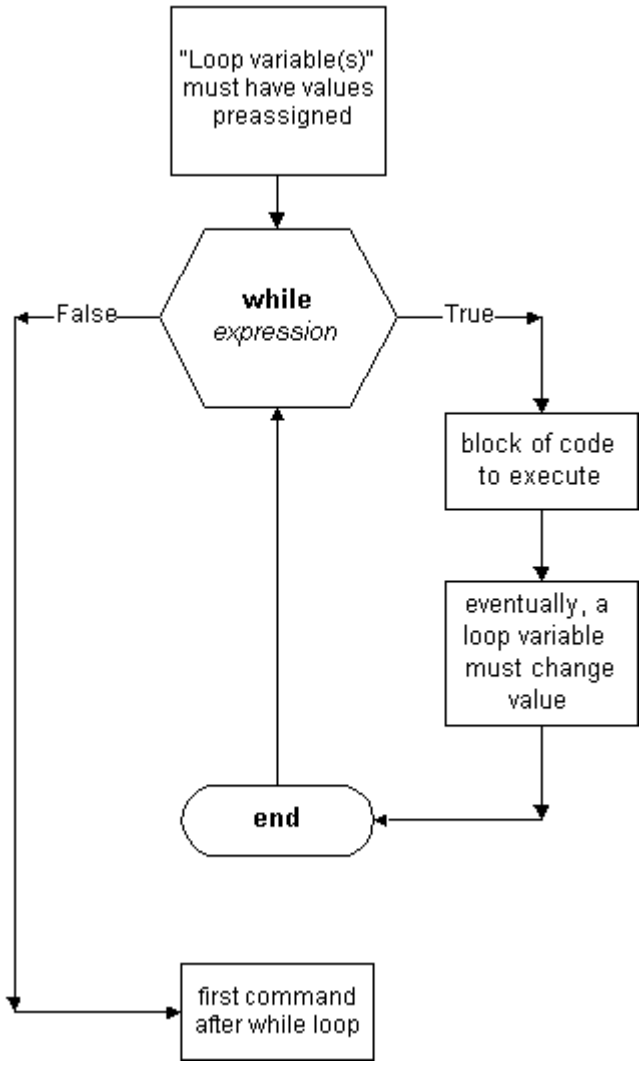
1. Basic "while" loop

Format of code:

```
while expression is true
     statements to be executed, also called the
     "body of the loop"
end
```

2. Flow chart

Figure 6. Flow chart of a basic "while" loop.



3. "while" Loop Rules

- Must have a variable defined BEFORE the "while" loop, so you can CHECK it to enter the loop.
- The variable in the while statement must change INSIDE the while loop, or you never exit the loop.
- After the loop is finished executing, the loop variable(s) still exist in memory and its value is the last value the variable had in the while loop.

Example 1: Basic "while" Loop.

- The loop variable, **x**, is defined before entering the loop.
- The value of x changes within the loop.
- When x = 120, the test x <= 100 fails, so the loop is exited.
- The loop executes four times.

```
x = 0;
while x <= 100
   x = x +30
end
```

```
x = 30
x = 60
x = 90
x = 120
```

Example 2: Use a "while" loop for interactive input.

- The loop variable, **flag**, is defined before entering the loop.
- Loop variable, **flag**, changes due by user input.
- The loop executes until the user enters a value of **flag** not equal to zero (while the instructions say to enter 1 if the values are correct, any nonzero value entered will cause flag = = 0 to be false).

```
flag = 0;
while flag == 0
   pres=input('Enter pressure: ')
   tmpk=input('Enter temperature in K: ')
   infile=input('Enter input filename: ','s')
   flag=input('Are these values correct? Answer 1 or 0, 1=yes, 0=no: ')
end
```

Example 3: Use a counter for a "while" loop.

- The loop variable, **i**, is defined before entering the loop.
- The value of **i** is incremented in the loop.
- Variables **x** and **y**, used in the loop, must be defined prior to entering the loop.
- This loop executes exactly 50 times. Since i =1 to start the loop, the test is for i <= 50.

```
x = 10;
y = 20;
i = 1;                    % Initialize the counter

while( i <= 50)
   x = x+(y^2)            % Calculate x
   i = i + 1              % Increment the counter
end
```

Example 4: Use a counter for a "while" loop; variation of Example 3.

- The loop variable, **i**, is defined as **zero** before entering the loop.
- The test is for i < 50.
- This loop executes exactly 50 times.

```
x = 10;
y = 20;
i = 0;                  % Start with i = 0

while( i < 50)          % New expression -- test for i < 50
   x = x+(y^2)          % Calculate x
   i = i+ 1             % Increment the counter
end
```

E. "break" COMMAND

- The purpose of the "break" command is to exit a "for" or "while" loop prior to the normal termination of the loop using the loop variable.
- Typical usage is for the "break" command to be issued within an "if" statement after a specified criterion is met.
  - The "if" statement is in the body of the loop, and exit is *immediate* from the loop.
  - The next MATLAB command executed is the first command after the "end" statement of the loop being exited.
- If the loops are nested and the break command is issued in the inner loop, the exit is from the inner loop, and execution continues in the outer loop.

Example 1: Break from inside one **while** loop.

```
i = 3;
j = 5;
while j <= 10
   z = (i^5) + 4*((j+2)^3);
   if z > 3500
      s=sprintf('Exiting loop,  i= %i  j= %i  z = %i  ',i,j,z);
      disp(s)
      break
   else
      s=sprintf('stay in loop, i= %i  j= %i  z = %i  ',i,j,z);
      disp(s)
   end
   j = j+1;
   i = i+1;
end
s=sprintf('Outside of While Loop, i =  %i',i); disp(s)
```

MATLAB Results:

```
stay in loop, i= 3  j= 5  z = 1615
stay in loop, i= 4  j= 6  z = 3072
Exiting loop,  i= 5  j= 7  z = 6041
Outside of While Loop, i =  5
```

Example 2: Break from inside the inner loop of two nested **while** loop.

```
i=5;
while i <= 8
   x = 2 + (i^4);
   j = 3;
   while j <= 10
     z = x + 4*((j+2)^3);
     if z > 3500
        s=sprintf('Exiting inner loop, i=%i j=%i z= %i ',i,j,z);
        disp(s)
        break
     else
        s=sprintf('stay in loop, i=%i  j=%i  z= %i ',i,j,z);
        disp(s)
     end
     j = j+1;
   end
   i = i+1;
end
s=sprintf('Outside of Outer Loop, i =  %i',i); disp(s)
```

MATLAB Results:

```
stay in loop, i=5  j=3  z= 1127
stay in loop, i=5  j=4  z= 1491
stay in loop, i=5  j=5  z= 1999
stay in loop, i=5  j=6  z= 2675
Exiting inner loop,  i=5  j=7  z= 3543
stay in loop, i=6  j=3  z= 1798
stay in loop, i=6  j=4  z= 2162
stay in loop, i=6  j=5  z= 2670
stay in loop, i=6  j=6  z= 3346
Exiting inner loop,  i=6  j=7  z= 4214
stay in loop, i=7  j=3  z= 2903
stay in loop, i=7  j=4  z= 3267
Exiting inner loop,  i=7  j=5  z= 3775
Exiting inner loop,  i=8  j=3  z= 4598
Outside of Outer Loop, i =  9
```

XI. FUNCTIONS

A. INTRODUCTION

Functions M-files are different from script M-files. Functions are blocks of code that need be executed many times. Functions can be executed in script M-file or by other functions. The following terminology is used in these notes:

- The command to execute a function is also referred to as a command "invoking a function" or "calling a function".
- A ***main program*** is typically a script M-file that calls one or more functions, but the term may also apply to a function that calls another function.

B. ADVANTAGES OF USING FUNCTIONS

- Break your program into separate tasks (modular programming).
- Write the code once, test it, and use it many times.
- The main program, which calls the function(s), is easier to read.
- Keeps the interactive MATLAB workspace free of unnecessary variables.
- Share functions with other programmers.

C. DISADVANTAGES OF USING FUNCTIONS

- Functions are harder to test because the local variables, which may be needed for debugging, are not passed to the interactive MATLAB workspace.
- Increases the number of M-files to keep track of in your directory.

D. REQUIRED ELEMENTS AND EXECUTION OF A FUNCTION M-FILE

1. The first line of ***function*** M-file has a mandatory structure. It has the word ***function*** in it, which identifies the file as a function, rather than a script M-file.

2. The first line contains the following elements, listed in the order they must appear in the line:

    - The word "function"
    - The output variables of the function enclosed in square brackets and separated by commas. If the function has no output variables, the square brackets are omitted.
    - The name of the function, which *exactly matches* the name of the file (function name is *abs_error*, and the file name is *abs_error.m*).
    - The input variables of the function enclosed in parentheses and separated by commas. If the function has no input variables, the parentheses are omitted.

3. For example, the first line of a function named *abs_error.m* is:

```
function [X,Y] = abs_error(A,B)
```

4. A block of comments should be placed at the top of the function M-file just ***after*** the function definition line. This is the header comment block includes the "H1" line and the comments to be viewed with the `help` command.

5. The "H1" line is the second line of the M-file, immediately after the function definition line. The "H1" line should contain keywords, which the built-in search function, "lookfor", uses. The "H1" line is not the same as the comment line describing the purpose of the function. However, the keywords in the "H1" line will also be used in the description of the purpose of the function.

6. Textbook section 4.2, p. 80-86, provides more details on functions, which are not included in these notes. Students are responsible for the information in this section of the textbook, *except* for the subsection on "feval" function, p. 85.

7. All function M-files for this course must have the following documentation information at the top of each file, as demonstrated in this example:

```
function [CHILL_C] = wchill(TC,WS_ms)
%  wind chill temperature, NWS formula
%  Purpose:  Calculate wind chill using new NWS formula
%
%  Input Variables:
%           TC    = Temperature (deg C)
%           WS_ms       = Wind Speed  (m/s)
%
%  Output Variables:
%           CHILL_C = Wind Chill Temperature(deg C)
%
%  Local Variables:
%           WS_kts  = Wind Speed (knots)
%           WS_mph  = Wind Speed  (mph)
%           TF      = Temperature (deg F)
%           CHILL_F = Wind Chill (deg F)
%
%  Functions Called:  None.
%
%  Reference: NWS, Service Change Notice 01-49, 30 Aug 2001
%
%  Filename: wchill.m
%  Written by: M. Jordan, MR 2020, 2/12/02

WS_kts = WS_ms*1.94386;    % Convert wind speed from m/s to knots
WS_mph = WS_kts*1.15078;   % Convert wind speed from knots to mph
TF = 32 + TC*(9/5);        % Convert TC to Fahrenheit
%      Calculate wind chill in Fahrenheit, using NWS formula
CHILL_F = 35.74 + 0.6215.*TF –35.75.*(WS_mph.^0.16) + …
          0.4275*TF.* (WS_mph.^0.16);
CHILL_C = (CHILL_F -32)*(5/9);  % Convert wind chill in F to C
```

E. FUNCTION VARIABLES: INPUT, OUTPUT AND LOCAL

- When a function executes, a separate area of memory is used for the function calculations. This memory is ***separate*** from the current MATLAB workspace and it is called the ***function workspace***.
  - For ***script*** M-files, the variables in the script program are stored in the ***current*** workspace.
  - For a ***function*** M-file, the input variables from the current workspace are added to the *function* workspace. All function commands are executed in the *function* workspace.
  - The input and output function variables are stored in the current workspace at the end of the function execution.
  - The memory used for function workspace is *cleared immediately* after the function finishes execution.

- **Input Variables** are variables from the current workspace, which are transferred to the function workspace.
    - The values of each input variable may be changed during the execution of the function. Typically, a programmer will not want to change the values of the input variables during the function execution, so be careful.
    - The input variables are re-stored from the function workspace to the current workspace at the end of the function execution.
- **Output Variables** are generated during the execution of the function and will be stored in the current workspace at the end of the function execution.
    - Only variables designated as output variables in the first line of the function (in square brackets) will be stored in the current workspace.
- **Local Variables** are variables in the function workspace, which are not designated as either input or output variables.
    - Local variables are erased from memory at the end of the function execution, and cannot be retrieved.
    - If a local variable is needed outside the function workspace, change the function output variable list to include the new variable. Also change all portions of the main program, which call the function and add the new variable to the output variable list.
- **Global Variables** are variables, which can be automatically shared between the current and function workspaces. Global variables have significant disadvantages, which outweigh the usefulness in METOC course and thesis programming. They are not covered in this course and are not recommended for use.

F. VARIABLES NAMES

- Since the current and function workspaces are separate, the variable names do not have to match.
    - If you are writing the function, use variable names that match those used in your main program or current workspace.
    - If you are using a function that someone else wrote, chances are the variable names will not match your naming conventions. This is not a problem.

- Using the function example for wind chill (in section D), in the function workspace the variables are named TC, WS_ms, CHILL_C.
    - If the same variable names are used in the current workspace, call the function with the command:

```
>>[CHILL_C] = wchill(TC,WS_ms)
```

    - If the variable names in the current workspace are TempC, Wspeed_ms, WindChill_C, call the function with the command:

```
>>[WindChill_C] = wchill(TempC,Wspeed_ms)
```

    - The result will be the same.
- MATLAB is passing the *memory location information* (not the variable name) to/from the function workspace, so the variable names do not matter.
- The *variable type, size, and units* must match between the current and function workspaces.
    - If the function is expecting a scalar variable, do not pass an array from the current workspace to the function.
    - If the function is expecting a column vector, don't pass a row vector.
    - If the function is written to use temperature array in degrees Celsius, do not pass it an array in degrees Fahrenheit.
    - The function is expecting the input and output variables in a specific order. If the function is expecting input variables, in order: Pressure, Temperature, Relative Humidity, do not pass the variables out of order.
        - MATLAB cannot detect variables passed out of order.
        - The result will be the wrong answer without any error messages.
    - For example, the first command is correct and the second command, with an incorrect variable sequence, will produce an erroneous error, without an error message.

```
>>[WindChill_C] = wchill(TempC,Wspeed_ms)   % correct

>>[WindChill_C] = wchill(Wspeed_ms,TempC)   % ERROR!!
```

- The same variables names may be used in the current and function workspaces, but have different meanings and values. MATLAB is not confused because only the memory location information is passed between workspaces.
    - This example emphasizes the need for thorough documentation of variable names and meanings in both a main program and function.
    - Documentation keeps the programmer from being confused!
    - Local variable names reside only in the function workspace.

Figure 7. Schematic to illustrate that variables names in the current and function workspaces are connected by passing memory location information, and not the variable name. Also illustrates that the variable names do not need to match between the current and function workspaces.

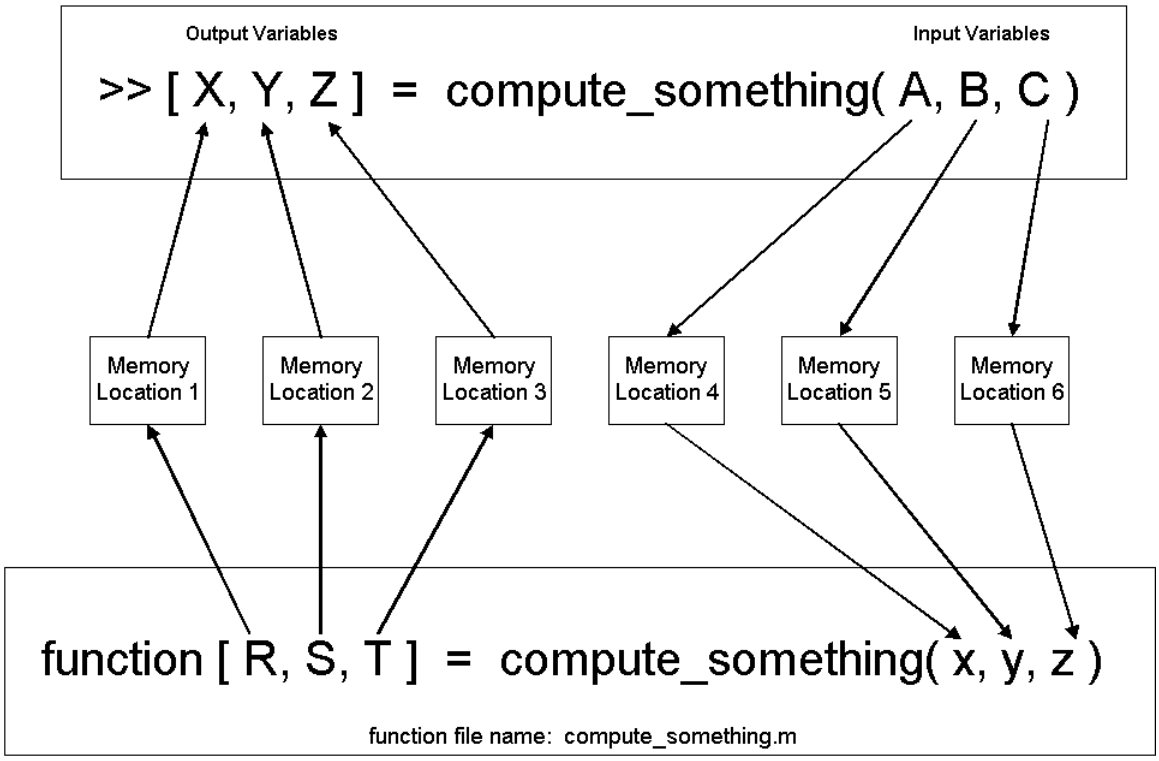
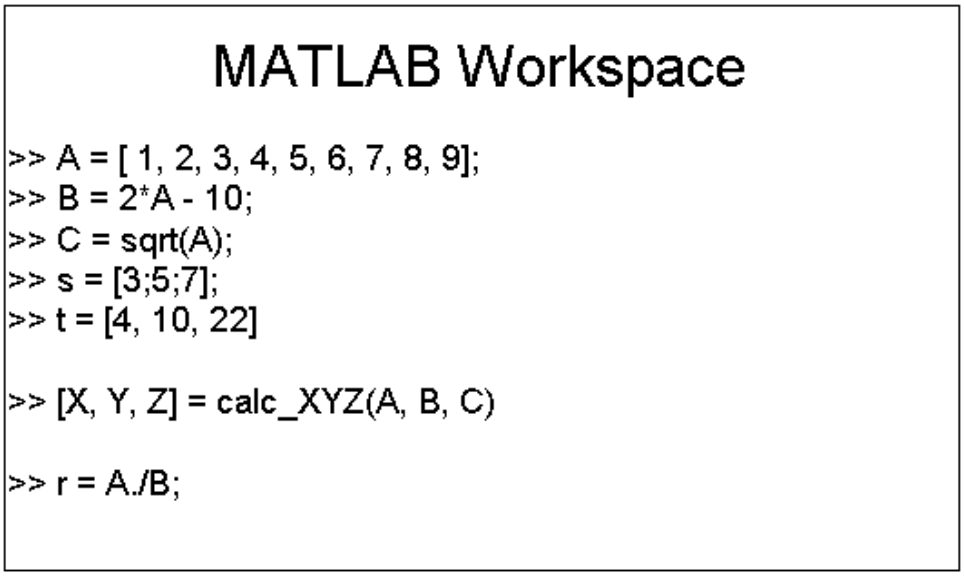
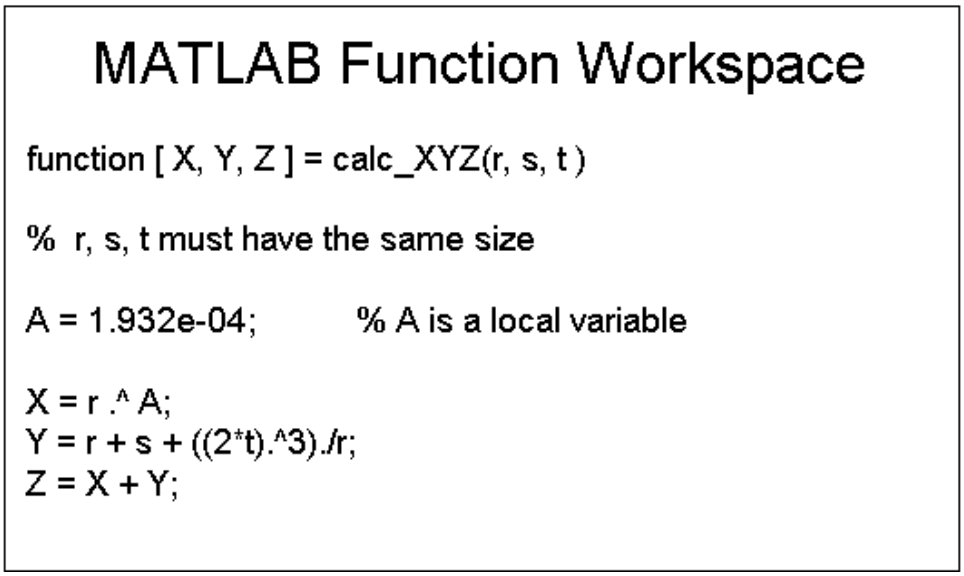


Figure 8. Schematic to illustrate the different contents of the current workspace (top, called "MATLAB Workspace") and the function workspace (bottom). Variables A, r, s, are used in both workspaces and have different array sizes and meanings. Thorough documentation is needed in the main program, which generated the current workspace, and the function, so the programmer is not confused by the duplicate variable names.





G. HOW TO WRITE AND TEST A FUNCTION

1. Define what the function will accomplish in pseudocode -- identify the steps.

2. Identify the input, output and local variables.

3. Choose meaningful variable names.

4. Write the lines of MATLAB code first as a *script* M-file (not as a function). The name of *this* M-file is not important.

5. In the interactive MATLAB workspace, test the *script* M-file.

   - Define input variables with test values.

   - The input variable names for the test *must* match the names used in the *script* M-file because all the variables are in the interactive workspace (and not "passed" to the function).

   - Run the script M-file. The results of the script M-file, output and local variables, are computed and reside in the current workspace.

   - Check that the steps and equations executed as *planned*. (The computer executes precisely what you program it to do, therefore you are checking if the MATLAB code performs the tasks you outlined in your pseudocode.)

   - If the equations use array operations, use arrays as your test variables.

6. Choose a meaningful function name (which is the name of the function M-file).

7. Open a text editor and assign the function filename to the file.

8. Write the first line of the function M-file following the correct syntax:

```
function [output variables] = filename(input variables)
```

9. Copy your code from the test script file into the function M-file.

10. Test your function M-file by calling the function from the interactive workspace. *For convenience*, use test input variable names that are the same as the names in the function. However, since you are now passing data to and from the function, the variable names do not have to match the variable names used in the function.

11. Document your function M-file using the guidelines described in section D. Also include:

    - For each variable: identify (1) units (mb, m/s, etc) and (2) type: scalar, array or matrix.

    - Limitations on use of the function. Add comments to remind yourself and other users of assumptions made which will limit the applicability of this code for other uses. Many times, the assumptions are due to the physics represented by the equations and constants.

H. THINGS TO CHECK WHEN CALLING A FUNCTION IN A PROGRAM

1. The function is in your current directory or in the MATLAB path.

2. Input variable lengths and types are appropriate for the function.

    - Don't input arrays if the program is written for scalars.
        - If variable X is written to be a scalar, don't input a vector or array.
        - If a function does not use array operators, the function will not work for arrays.
    - Arrays must be the correct size and shape.
    - Units of the input data match the units used in the function.

3. Input variables in the call statement are listed in the same order as the function. The variable names are not required to match, but the type of data (temperature, etc) must match what the function expects.

4. Output variables in the call statement are listed in the correct order; variable names are not required to match.

5. Main program statement, which calls the function, has the correct syntax.

XII. GENERAL PROGRAMMING TIPS

A. VARIABLE NAMES

- Variables should have meaningful names. This will make your code easier to read, and will reduce the number of comments you will need. However here are some pitfalls about choosing variable names:
  - Meaningful variable names are good, but when the variable name gets to 15 characters or more, it tends to obscure rather than improve code.
  - The maximum length of a variable name is 31 characters and all variables *must start with a character (not a number)*.
  - Be careful of naming a variable that will conflict with MATLAB built-in functions, or reserved names: if, for, while, end, pi, sin, cos, etc.
  - Avoid names that differ only in case, look similar, or differ only slightly from each other.

B. M-FILES

- Make good use of white space, both horizontally and vertically, it will improve the readability of your program greatly.
- Comments describing tricky parts of the code, assumptions, or design decisions should be placed above the part of the code you are attempting to document.
- Do not add comment statements to explain things that are obvious.
- Try to avoid big blocks of comments except in the detailed description of the M-file in the header block.
- Indent lines of code and comments inside branching (if block) or repeating (for and while loop) logic structures will be indented 3 spaces.
  - Please note: *don't use tabs, use spaces*.
- No more than one executable statement per line in your script or function M-files.
- No line of code should exceed 80 characters. (There may be a few times when this is not possible, but they are rare.) Use the ellipse (**…**) to continue your command on the following line.

- Be careful what numbers you "hardwire" into your program. You may want to assign a constant number to a variable. If you need to change the value of the constant before you re-run the program, you can change the number in one place, rather than searching throughout your program.
    - For example:

```
%  This program "hardwires" the constant 100
%  in two places in the code.
for i = 1:100
   data(i) = r(i)^2;
end
meanData = sum(data)/100;

-----------------------------------------------------
%  This program assigns the constant value, 100,
%  to the variable, n.
n = 100;              % number of data points.
for i = 1:n
   data(i) = r(i)^2;
end
meanData = sum(data)/n;
```

XIII. PLOTTING WITH MATLAB

A. BASIC PLOT COMMANDS

MATLAB has a versatile plotting package and excellent on-line help. The MR 2020 textbook, "*Getting Started with MATLAB 5*", R. Pratap, 2nd Ed, 1999, has an excellent overview and lots of examples of 2-D and 3-D plots. However, the textbook is written for MATLAB 5.0-5.2, and there are options of MATLAB 5.3 and 6.0 that the textbook does not cover. *If in doubt, use the on-line help to augment the textbook discussion.*

There are several options for accessing **on-line help**.

- The help command (started interactively) Interactively, will list the help in the MATLAB command window.

  ```
  >> help command
  ```

- The MATLAB *helpdesk* (started interactively) allows for easy access to help for plotting commands using an Internet browser.

  ```
  >> helpdesk
  ```

- Both methods list related commands in the "see also" section at the end of the help discussion. This is a very useful feature.

The instructor-provided M-files of plotting commands used in the lab assignments provide examples of how to use common 2-D plot commands. Three-dimensional plots of meteorological data are typically generated in VIS5D, not MATLAB, so 3-D plotting is not taught in MR 2020. The textbook covers the basics of 3-D plots.

Students should review the following sections of the textbook and, as needed, on-line help for the commands and topics listed in Tables 3 and 4.

Table 3. Basic Plot Commands

MATLAB Command or Topic	**Textbook Reference or Helpdesk: "Using MATLAB Graphics" topic**
plot plot(Temp,Ht,'r--*') plot(Temp,Ht,'b-',DewPt,Ht,'r- -')	Sections 6.1, 6.1.1. and 6.1.5
hold on / hold off	Section 6.5 and on-line help
line	Section 6.5
subplot (multiple plots)	Section 6.2
figure figure(*number*)	See on-line help
xlabel	Section 6.1.2
ylabel	Section 6.1.2
title	Section 6.1.2
text	Section 6.1.2
gtext	Section 6.1.2
legend	Section 6.1.2
axis axis ij flips vertical axis	Section 6.1.3 -- see on-line help for newer syntax than in textbook
plotedit	Section 6.1.4 Helpdesk: Search for function "plotedit"
propedit	Sections 6.1.4 and 6.4
close close all	See on-line help
clf … clears current figure	See on-line help
Greek symbols in text strings ' \pi \Pi '	Helpdesk: Search for "Greek letters"

Table 4.Advanced plotting options: 2-D and 3-D plots.

MATLAB Command or Topic	**Textbook Reference or Helpdesk: "Using MATLAB Graphics" topic**
Specialized 2-D plots	Section 6.1.6, p. 154-159 help graph2d help specgraph
3-D plots	Section 6.3 help graph3d
plot3	Section 6.3
comet3	Section 6.3
view	Section 6.3.1
Mesh and surface plots Function: meshgrid	Section 6.3.3
Interpolated surface plots Function: griddata	Section 6.3.4

B. HANDLE GRAPHICS

A full discussion of handle graphics is too detailed for this course. The basic plotting commands are fine for most homework assignments. A brief description is presented of ways to upgrade the plot quality for formal presentations and thesis figures.

The most common plot elements to change or improve are: line thickness, line color, grid spacing, grid labels, axis label and title fonts, adding Greek symbols.

There are two ways to accomplish these improvements: (1) interactively using menus activated in each figure window, and (2) using handle graphics commands in M-files. If you are making changes to 1-5 plots, you may want to do it interactively. For plotting style consistency when generating many figures and thesis figures, putting handle graphics commands in M-files is the preferred method.

Interactive Method: Activate the **Plot Edit** option on the figure window (click the 5th icon in the figure toolbar, or choose help). Once plot edit is activated, the Tools menu in the figure window provides menus to change properties: axis, line and text, and to add: arrow, line, text. Alternately, click the left mouse button on any line or text and then use the right mouse button to activate a menu of properties for that line/text.

Handle Graphics Commands: The textbook Section 6.4 covers handle graphics and provides basic examples (p. 180-183). The helpdesk topic on Handle Graphics is quite extensive and may be too advanced for many users.

Using PowerPoint to change title and axis labels: To make title more readable or to add Greek symbols (other than the method in Table 3), the easiest way to do it may be in PowerPoint. Print the figure without title, xlabel and ylabel. Insert the figure into PowerPoint and add text strings using PowerPoint commands. Group the text and figure together (PowerPoint option). This new figure can be copied into Word or another word processor.

Example: A sample program with basic handle graphics commands is included.

- **Figure 9** is a plot without any handle graphics changes.
- **Figure 10** was plotted with the program, `sample_handle_graphics.m`, listed below, which demonstrates basic handle graphics commands.
- Read Section 6.4 to understand the basics of handle graphics and then review the sample program.
- Most METOC students will not need more sophisticated commands than the ones listed here.

Figure 9. MATLAB plot created without handle graphics commands.

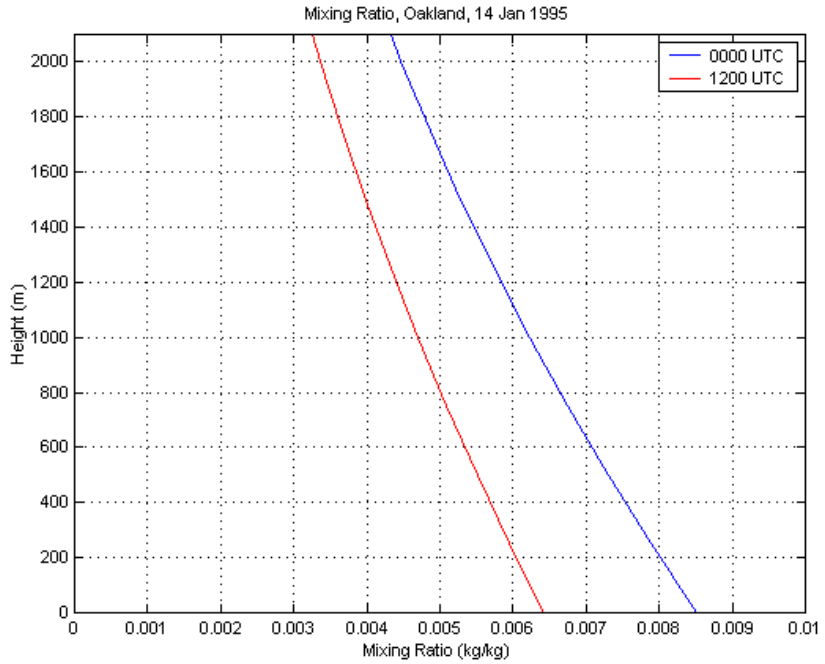
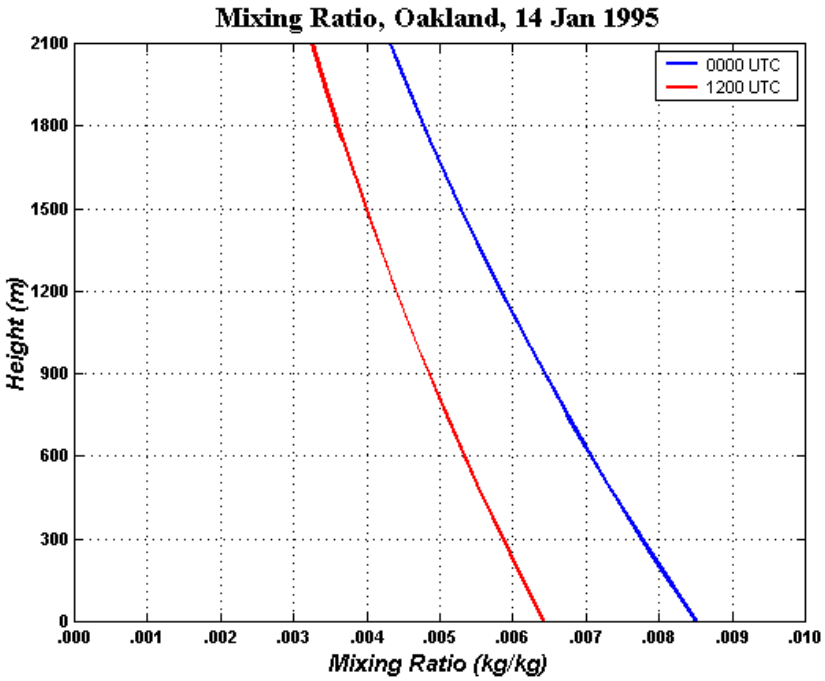


Figure 10. MATLAB plot created with handle graphics commands, using the program, *sample_handle_graphics.m*.



```
% file = sample_handle_graphics.m  M. Jordan, 2/27/2001, for MR 2020

load handout3.mat     % data file

figure
plot(mix_ratio1,Z,'b','LineWidth',2)  % change line width in the plot command
hold on
plot(mix_ratio2,Z,'r','LineWidth',2)  % change line width in the plot command
grid on
axis([0 .01 0 2100])
legend('0000 UTC','1200 UTC')   % must change "parent" properties to
                                 % change the legend font
%   ... change axis information .. handle is already known:  "gca"
%       change the XTick and YTick marks

set(gca,'XTick',[0:.001:.01])
set(gca,'YTick',[0:300:2100])

%   ... change Labels of the XTick marks .. handle is already known:  "gca"

%       Step 1.  make an Array of equal length character strings which
%       will be the new labels

xaxis_label = ['.000';'.001';'.002';'.003';'.004';'.005'; ...
               '.006';'.007';'.008';'.009';'.010'];

%       Step 2.  change the XTickLabel to the strings in xaxis_label

set(gca,'XTickLabel',xaxis_label)

%   ... change the font size/weight for the axes
%       NOTE: this sets the font for both x and y axes:
%             xlabel, ylabel and numbers on each axis

set(gca,'FontSize',10,'FontWeight','bold')

%   ... add legend, title, x and ylabels while allowing the handle information
%       (output of the function) to be stored in the defined variable names

handle_title  = title('Mixing Ratio, Oakland, 14 Jan 1995')
handle_xlabel = xlabel('Mixing Ratio (kg/kg)')
handle_ylabel = ylabel('Height (m)')

%   ... change the fonts for title, and x, y labels
%       overwrites the fonts for xlabel, ylabel defined above (10 pt size)

set(handle_title,'FontName','times','FontSize',16)

set(handle_xlabel,'FontName','arial','FontSize',14,'FontAngle','italic')

set(handle_ylabel,'FontName','arial','FontSize',14,'FontAngle','italic')
```

C. PLOTTING A COASTLINE

Obtain Coastline Data: use the Worldwide Coastline Extractor from NGDC

- Download the coastline at: **http://rimmer.ngdc.noaa.gov/coast/**
- For an input latitude and longitude range, extracts the coastline lat/lon.
- Makes an ASCII text file, readable by MATLAB, of longitude and latitude, which defines the coastline.
    - Must choose the MATLAB format option
- Several coastline databases options.
- Download the ASCII file and rename it with a meaningful name.

Import the Coastline to MATLAB

- Load the ASCII file into MATLAB. Break the data into column vectors
    - Longitude is Column 1
    - Latitude is Column 2
- Save the lat/lon variables as a MATLAB ".mat" file, for future use.
- Plot the coastline
    - **>> plot(longitude,latitude)**
    - Use the axis command to rescale the axes, as necessary.
    - West longitude is a negative number.

How to Download the Coastline

- **http://rimmer.ngdc.noaa.gov/coast/**
- Choose the latitude and longitude range.
- Choose the coastline database (1:70,000 has higher resolution than 1:250,000).
- Choose the MATLAB format option.
- The GUI options are similar to (partial list):

Coastline data base:

NOAA/NOS Medium Resolution Coastline (designed for 1:70,000)

Compression method for extracted ASCII data:

(•) None ( ) GNU GZIP ( ) UNIX Compressed ( ) ZIP

Coast Format options:

( ) Mapgen ( ) Arc/Info Ungenerate (•) Matlab ( ) Splus

Coast Preview options:

( ) No preview (•) GMT Plot

D. PRINTING AND SAVING A DIGITAL IMAGE

There are several printing options:

- Print directly to a printer from the command prompt.
- Make a postscript file (color or B&W) and save it on a disk to print later.
- Make a digital image to import into MS Word or PowerPoint.

1. Print from the command prompt.

```
>> print          % prints to the attached printer
```

- As a general rule, ***don't put the print command in the M-file***.
- Many times, when you issue the print command inside an M-file later you want to make one more change and must reprint the figure.
- Save paper, print interactively at the command prompt.
- Use the **`figure(number)`** command to set the pointer to the desired figure window.
- At the command prompt, type **`print`**

2. To save a MATLAB plot as a postscript *file*

Use the print command (at command prompt or in M-file), with options:

% prints as Black lines on White background

```
>> print -dps filename.ps     % ".ps" indicates postscript
```

% prints as Color lines on white background

```
>> print -dpsc filename.psc   % ".psc" indicates color postscript
```

3. Printing a *postscript file*

- The IDEA Lab printer is a B&W postscript printer.
- ***For most class projects and homework, B&W is fine.***
- To print out the postscript file in the IDEA Lab, the UNIX command is:

  `lp filename.ps`

- The Graphics Lab has a color postscript printer, but it is expensive to use.

- To print out the Color postscript file in the Graphics Lab, you MUST be logged into a Graphics Lab workstation. The UNIX command is:

```
lp -dcolor filename.psc
```

- Note: If you print a color postscript file to a B&W printer, the output is poor. The color postscript files print in shades of gray and not as black lines. Therefore, a red line in the color plot prints as light gray, and is hard to see.

4. To save a MATLAB plot as a graphics *file*:

If you need a plot to e-mail to someone, import into a word processor, or PowerPoint, you need the figure in a graphics format.

- See on-line **>>help print** for the many graphics options.
- "gif" is not supported by MATLAB.
- Experience has shown the best images are produced by converting the image to **tiff**. This produces an image wider that 8.5 inches.

```
>> print –dtiff  filename.tif
```

- To make a **tiff** image sized to 6.0 inches (fits a MS Word page), use the "-r75" option. A factor larger than 75 makes the image larger, smaller factor reduces the size.

```
>> print –dtiff  -r75  filename.tif
```

5. Convert image formats using "xv".

To convert from one graphics format to another, use the UNIX software **xv**.

- Use the UNIX program **xv**. At a UNIX prompt, type **xv**
- Click the RIGHT mouse button to get the menu.
- Click on LOAD.
- Select the gif, tif, pcx, postscript, or other digital file from the list.
- Click on SAVE.
- Click on FORMAT and select the format to "save as".
- Choose any of the popular formats needed for other graphics programs.

- It is also possible to convert a graphics file into postscript. When you save as a postscript file, be sure to size the image to fit on 8 1/2 x 11 paper.
- Another option is to use "xv" to view postscript files.
- **Never** use the ***xv menu*** to ***print anything*** (gif, tif, jpeg, postscript, etc).
    - Use UNIX to print the postscript file.
    - Printing from **xv** itself causes the printer to print the file as ASCII instead of the digital format. Consequently *hundreds of sheets* of paper are generated and wasted.

XIV. HOW TO USE "FTP"

A. OVERVIEW

File transfer protocol (ftp) is one method used to remotely login to another machine and transfer files. Only computers that have been set up for remote login will allow ftp access. Before trying to ftp to another machine, you must know/have the following:

- Network name or IP address of the remote machine.
- User account and password on the remote machine.
- Ftp software must be available on computer you are using.

B. PROCEDURES TO FTP TO THE IDEA/GRAPHICS LABS

1. Log into an NPS Network PC with your user name/password.
2. Start WS-FTP software on the Network PC (if not on your Desktop, look under Programs).

    - Select NEW (sets inputs to blank)
    - Enter an IDEA Lab or Graphics Lab computer name:

      Fastest way to access Meteorology Department Disks:

      mojave.met.nps.navy.mil or **eagle.met.nps.navy.mil**

      or use any IDEA Lab UNIX workstation name (lower case) and **.met.nps.navy.mil**

      Fastest way to access Oceanography Department Disks:

      attu.oc.nps.navy.mil or **tonga.oc.nps.navy.mil**

      or any Graphics Lab UNIX workstation name (lower case) and **.oc.nps.navy.mil**

    - Enter your IDEA Lab account name
    - Enter **OKAY**
    - **DO NOT** enter your password and **store** it on the computer
    - Enter your password when prompted, then ENTER.
    - Wait for the connection to be made.

- A GUI interface will appear.

    i. Left side is the HOST computer (Network PC machine)
    ii. Right side is the REMOTE UNIX computer pointing at your **home directory**
    iii. Use **Chdir** button to **Change directory** on both the Host and Remote computers.

        - On the Host computer, use your H: drive or floppy disk or Zip100 disk. Don't write to the Network machine's C: drive.

    iv. Highlight the file you want to transfer.
    v. Select ASCII or BINARY

        - ASCII text files, ".m" files should be transferred in ASCII mode
        - Graphics files, ".mat", MSWord, PowerPoint, Excel must be transferred in BINARY mode

    vi. Push the arrow button to transfer the file.
    vii. Multiple files may be highlighted and transferred at once.
    viii. File transfer is permitted in either direction.

- **Ending a session** is a *two step* process:

    i. Use the Close button (left button on the bottom bar)
    ii. Use the Exit button (right button on the bottom bar)

C. FTP TO AN NPS NETWORK PC

- The computer name and/or IP address for the Network PCs are not apparent. Therefore, unless you have an address provided by an IT consultant, you cannot ftp ***to*** an Network PC ***from*** the IDEA or Graphics Labs.

D. FTP TO THE IDEA OR GRAPHICS LABS FROM OFF-CAMPUS

- Due to changes in the NPS firewall, students must consult the IT Helpdesk for this information.

XV. CHARACTER STRINGS

A. RULES FOR CHARACTER STRINGS

1. Character strings are defined with single quotes (') around a string of alphanumeric characters.

2. There are many uses for character strings, such as input to plotting functions: *title*, *xlabel*, and *ylabel*.

3. One character string, with N alphanumeric characters, is stored as a row vector with size: 1xN.

4. The character string may be assigned to a variable name, but it is not required. Character string variable names follow the same MATLAB rules for variable names.

5. The list generated by the *whos* command identifies which variables are character strings.

6. Individual character strings can be ***concatenated*** together to form a new, longer string.

7. Multiple character strings can be stored together in a *matrix*.

8. An apostrophe or single quote may be included in a character string by using *two single quotes together* in the string.

```
>> 'Sally''s Plot for MR 2020'

ans =
Sally's Plot for MR 2020
```

9. An *advanced* use of character strings is to concatenate a character string to be a MATLAB command, such as 'load sounding_data.mat', and to execute the string as a command by using the ***eval*** function: **eval('load sounding_data.mat')**.

B. CHARACTER STRINGS AS ROW VECTORS

1. One character string, with N characters, is stored as a row vector, size 1 x N.

```
>>s = 'This is a character string'  % 26 chars, including blanks
```

2. To extract characters 6-14:

```
>>t = s(6:14)
  t =
      is a char
```

3. The transpose of a row vector, **s'**, is defined, but a column vector of characters isn't useful.

C. CONCATENATION

- Multiple character strings can be concatenated to form a new, longer string. This is a very useful feature.
- Row vectors are concatenated using the same rules as for vectors of numbers.

```
a = 'Today is:'
b = '10 January'

r = [a, b]          % the comma is optional; r = [a b]

r =
    Today is:10 January

s = [a, ' ', b]     % a blank space is added between the
                    % two strings
s =
    Today is: 10 January

s2 = [a ' ' b]      % no commas between strings;
                    % same result.
s =
    Today is: 10 January
```

- Portions of strings may be concatenated:

```
t = [a(1:5) b(1:5)]
t =
    Today10Jan
```

D. CONVERT NUMBERS TO CHARACTER STRINGS

Two functions ***num2str*** and ***int2str*** are extremely useful. They convert real numbers and integers into character strings. This is useful when concatenation character strings.

- **num2str** converts a *real number* into a character string.

```
x = 16.35                                  % real number

u = ['The value of x is: ' num2str(x)]  % added to a string
```

```
u =
    The value of x is: 16.35
```

- **int2str** takes a real number and rounds it to the nearest integer and the integer is converted into a string.

```
x = 16.55                                % real number

v = ['The integer value of x is: ' int2str(x)]

v =
    The integer value of x is: 16
```

E. MATRICES OF CHARACTER STRINGS

- Multiple character strings may be stored in a **matrix** provided that ***all strings have the same length***.
  - Shorter strings can be stored in a matrix only if blank spaces are added to the end of the string (called ***padding*** the string) to make it the same length as the other strings in the matrix.
  - For example, if a matrix of names already contained 'David' and 'Kevin', the name 'Joe' could be added only after padding the string with two blank characters: 'Joe '.
- The function ***char*** will take input strings of different lengths, pad spaces where necessary, and make a matrix of same-length strings.
- Matrices of equal-length strings can be defined using ***square brackets*** and a ***semicolon*** between names:
- **Example 1:** Defining a matrix of ***equal-length*** character strings.

```
>>names = ['Mike'; 'Mary'; 'Jeff'; 'Jose'; 'Gary']
  names =
          Mike
          Mary
          Jeff
          Jose
          Gary
```

Note: the size of names is 5 x 4 -- 5 rows, 4 columns for each name.

- **Example 2:** Use the **char** function to create a matrix with *equal-length* strings when the *input strings are not the same length*.
  - Since the **char** is a function, a ***comma*** separates the *input* strings:

```
>>N = char('Joe', 'Sally', 'Margaret')
  N =
      Joe
      Sally
      Margaret
```

Note: matrix size is 3x8 -- the first two names have padded blank spaces.

F. INDIVIDUAL CHARACTER STRINGS IN A MATRIX

- To use individual character strings in a matrix, you must explicitly state the ***rows and columns*** of the matrix element.
- A person would look at a matrix of character strings, for example the matrix **names**, defined in Example 1 above, and think the matrix only has 5 elements.
  - Our intuition is view the element **names(1)** to be 'Mike'. This is not the case.
  - The matrix **names** is a 5x4 matrix. The element **names(1)** is only one character: 'M'.
- The string 'Mike' is stored in the *first row* and *all columns* of the matrix **names**. The string 'Mike' is stored in **`names(1,:)`**.
- To indicate the ***entire*** string in a matrix, *use the colon to indicate all columns of a row*.

```
>>a = names(3,:)

  a =
      Jeff
```

- To indicate ***partial strings***, use the *colon* and two numbers (e.g. 3:9) to indicate which columns.

```
>>b = names(1,2:4)

  b =
     ike
```

- More than one string can be extracted from a matrix by using the colon and two numbers to indicate which rows of the matrix.

```
>>c = names(3:5,:)

  c =
      Jeff
      Jose
      Gary
```

G. ADDING STRINGS TO EXISTING MATRICES

- There are some subtle aspects to adding strings to existing matrices. One character string is stored in one row and multiple columns.

- To add the ***first string*** to a ***new*** matrix, Y, this command works:

```
Y = 'string number 1'          % Correct
```

- ***Intuition*** suggests that the following command would work. *It does not!*

```
Y(1) = 'string number 1'       % Wrong
```

- The ***columns*** of Y(1) must be identified ***using the colon***. This is the command that works:

```
Y(1,:) =  'string number 1'    % Correct
```

- To add a ***second*** string to row number two, the ***columns*** of Y(2) must be identified ***using the colon***. This is the command that works:

```
Y(2,:) =  'string number 2'    % Correct
```

- To summarize:
  - Assigning one string to a variable can be accomplished with a *shorthand* command: Z = 'string1'.
  - To assign the second and subsequent strings to the matrix requires that the colon be used to signify "*all columns*". This method works ***only if*** all character strings are the *same length*.
  - To assign strings of different length to the same matrix, use the function ***char***, which will pad the strings to become the same length as the longest input string.

H. "for" LOOP TO CREATE A MATRIX OF CHARACTER STRINGS

- Using the loop variable *i* and the *int2str or num2str* function is a handy way of building a matrix of character strings with numbers concatenated.
- **Example 1:** Consecutive number in the strings (same number of digits in each number).

```
for I = 1:3
   Y(I,:) = ['string number ' int2str(I)];
end

Y =
string number 1
string number 2
string number 3
```

- Example 2: Non-consecutive numbers in the strings (same number of digits in each number).

```
Z = [14 77 99];    % same number of digits
                   % in each number

for I = 1:length(Z)
   Y(I,:)=['string' int2str(Z(I))];  % number is Z(I)
end

Y =
string14
string77
string99
```

- Example 3: Non-consecutive numbers in the strings (***different*** number of digits in each number). Use ***char*** function to add blanks.

```
Z = [1 77 100];  % different number of digits

for I = 1:length(Z)

   if I == 1
      Y = ['string' int2str(Z(I))];      % store 1st string
   else
      Y = char(Y,['string' int2str(Z(I))]);   % store as Y
   end

end

Y =
   string1
   string77
   string100
```

XVI. "eval" FUNCTION

A. OVERVIEW

- This function will execute (or ***eval***uate) strings containing MATLAB expressions. Usage:

  `eval('`*`string`*`')`

- A programmer will recognize that the variable, Y, is a character string that contains a command, Y='load mydata.mat', but MATLAB interprets Y as a character string, size 1x15.

- When the character string is the input to a function called eval, then MATLAB will interpret the string as a valid command and execute the command. The following commands produce the same result

```
>>Y = 'load mydata.mat'
>>eval(Y)                  % option 1
>>eval('load mydata.mat')  % option 2
```

- Typically, a command is concatenated from multiple character strings and may only be executed using the ***eval*** function.

- After evaluating an expression, there may be output data to be stored. The ***eval*** function allows for output. The data loaded from the data file will be stored in variable "data".

```
>>data = eval('load OAK_snd_13Feb02.txt')
```

B. EXAMPLES

- **Load a file**. The output variable, "data", will contain the sounding data in the sounding file.

```
>>data = eval('load oak_sounding_20011231_12z.txt')
```

- **Concatenate a ".mat" filename and save variables.**

```
>>date = 'oct25';
>>time = '00Z';

>>filename = ['OAK_' date '_' time '.mat']; % make string

>>eval(['save ' filename 'T Pres RH'])  % execute command
```

- **Concatenate a filename in nested "for" loops.** Use the eval command to load the file.

```
date=['oct25';'oct26';'oct27';'oct28'];
time=['00Z'; '12Z'];

for i = 1:4
   for j = 1:2

      filename = ['OAK_' date(i,:) '_' time(j,:) '.mat'];
      eval(['load ' filename])
      % add other commands which use the loaded data
   end
end
```

- **Concatenate a variable name.** Make a matrix with a new variable name from a character string containing the name of a previously defined variable.

  - Assume these variables exist in your MATLAB workspace:

```
>>whos
    OAK_12z    57x8       double array
    OAK_18z    89x8       double array
```

  - Use a "for" loop to concatenate the name of an existing variable, and store the *string name* in variable 'name'.

```
for I = [12 18]
    name = ['OAK_' num2str(I) 'z']
end
```

  - The variable, 'name', is a 1x7 character string and does not contain the data in the matrices `OAK_12z`, `OAK_18z`. Use the ***eval*** command to convert 'name' into the matrix of data.

```
for I = [12 18]
    name = ['OAK_' num2str(I) 'z'];
    data = eval(name);            % matrix
    hght = data(:,1);
    relh = data(:,2);
    figure
    plot(relh,hght)
    title(['Oakland Sounding ' num2str(I) 'Z'])
end
```

XVII. SEARCHING DATA ARRAYS AND MATRICES

A. OVERVIEW

There are times a programmer must search a data array or matrix and identify and extract a ***subset*** of the original data values.

- There is a "long" way to accomplish this task, using a "for" loop and "if" statement.
- MATLAB has two "shortcut" methods to search a vector or matrix.
    1. Using the "find" function.
    2. Using relational and logical operators.

B. BASIC CONCEPTS AND DEFINITIONS

To understand the two MATLAB shortcuts, five definitions and basic concepts must be explained:

1. Array Index
2. Matrix Index
3. "find" Command
4. Logical Arrays and Matrices
5. Relational and Logical Operators using Arrays and Matrices

C. ARRAY INDEX

Individual array elements are accessed by using the array index, or location. Multiple elements (a subset) of one array may be accessed by using another array, which contains indices (locations) of the first matrix.

- Given an ***array***, M = [10 20 30 40 50 60 70 80 90 100]
- Define a subset of M, let subset S = {10, 40, 90}
- The **indices** of M that correspond to the subset, S, are {1, 4, 9}.

- A variable may be defined to contain array indices and is used to extract a subset from the original matrix.

```
>> x = [1, 4, 9]          % an array of indices of M
>> S = M(x)               % S is a subset of M

   S =
  10    40    90
```

- Rules:
  1. The data array can be a row or column vector.
  2. The indices variable can be either a row or column vector and will work with the data array, regardless of its orientation (in the above example, if x is a column vector, it will still work with row vector, M).
  3. The orientation of the subset, whether a row or column vector, is determined by the orientation of the original array.

D. MATRIX INDEX

- Individual **matrix** elements are usually accessed by a row and column *pair* of indices (eg. A(5,12)).
- The matrix elements may also be addressed by a single number index.
- Matrix locations are numbered *columnwise*.
- An array of numbers, referring to locations in a data matrix, may be used as indices to extract a subset from the original matrix.
- Given an ***matrix***, A

```
A =
   100   200   300
   400   500   600
   700   800   900
```

Indices of A are numbered *columnwise*

```
   1   4   7
   2   5   8
   3   6   9
```

- Define a subset of A, let subset S = {100, 300, 500, 700, 900}
- The **indices** of A that correspond to S are {1, 7, 5, 3, 9}, respectively.

- The subset of A will be either a row or column *vector*. The orientation of the subset array is determined by the orientation of the subset indices array.

```
>> y = [1, 7, 5, 3, 9]          % an array of indices of A
>> B = A(y)                     % B is a subset of A
B =
    100    300    500    700    900

>> y2 = [1; 7; 5; 3; 9]         % a column vector of indices
>> B2 = A(y2)

B2 =
   100
   300
   500
   700
   900
```

- The order in which the elements of A are listed in the subset vector is determined by the order in which the indices are listed in the indices array.

```
>> y3 = [9 1 7 5 3]     % indices in nonsequential order
>> B3 = A(y3)

B3 =
   900   100   300   500   700
```

E. "find" FUNCTION

The find function searches an input *array or matrix*, for input search criteria, and returns the indices of array or matrix elements that meet the criteria. The indices are automatically listed in ascending order.

- Example with a data **array**. The orientation of the subset will match the orientation of the data row or column vector.

```
>> T = [10 15 18 22 13]          % array of temperatures
>> c = find(T >= 18)             % output are indices of T

c =
        3     4

>> T_gt_18 = T(c)                % subset T >= 18

T_gt_18 =
       18    22
```

- Example with a data **matrix**. The orientation of the subset will be a column vector.

```
>> Temp = [10 15 18; 22 13 6; 0 17 5]          % matrix
Temp =
    10    15    18
    22    13     6
     0    17     5

>> d = find(Temp >= 12)          % output are indices of Temp
d =
     2
     4
     5
     6
     7

>> Temp_gt_12 = Temp(d)          % Subset Temp >= 12

Temp_gt_12 =
         22
         15
         13
         17
         18
```

- The ***default*** use of the ***find*** function is to find ***nonzero*** values in an array/matrix.

```
>> A = [-2 0 2 0 4]
>> B = find(A)                   % finds nonzero values of A

B =
     1
     3
     5
```

F. LOGICAL ARRAYS AND MATRICES

- Logical arrays and matrices are arrays/matrices of 1's and 0's that can be used for logical indexing or logical tests.
- Logical arrays are also created by the relational operators (==,<,>,~, etc.) and MATLAB built-in functions ***any***, ***all***, ***isnan***, ***isinf***, and ***isfinite***.

- There is a difference between an array/matrix of the *numbers* 1 and 0 and an array/matrix that is defined to be a logical variable.
    - The ***whos*** command will indicate if a variable contains real numbers or is a logical variable.
    - The built-in function ***logical*** converts an array/matrix of 1's and 0's from real numbers to a logical variable.
- If an array/matrix is a logical variable, then it can be used to extract a subset of another array/matrix.
- If the logical value is 1 (true), the element is extracted, otherwise the element is omitted from the subset. ***The logical array must have the same number of element as the data array.***
- Example of logical **array**. Orientation of the output matches the data array.

```
>> T = [10 15 18 22 13]       % array of temperatures

>> f = [1 0 0 1 0]            % array of numbers (not logical)

>> g = logical(f)             % convert to a logical variable

>> T_subset = T(g)            % subset of T

T_subset =
   10    22

>> whos
      T             1x5            double array
      T_subset      1x2            double array
      f             1x5            double array
      g             1x5            double array (logical)
```

- Example of logical **matrix**. Output is a column vector.

```
>> Temp = [10 15 18; 22 13 6; 0 17 5]     % matrix

>> h = [0 1 0;1 1 0; 0 0 1]               % numbers not logical

>> k = logical(h)                         % convert to logical

>> Temp_subset = Temp(k)                  % subset of temp

Temp_subset =
      22
      15
      13
      5
```

G. RELATIONAL AND LOGICAL OPERATORS USING ARRAYS AND MATRICES

The rules of using relational and logical operators on scalar variables apply to arrays and matrices. The ***output*** of applying an expression using relational and logical operators to an array or matrix is a ***logical variable***.

- **Example 1: Search one array.** Output has the same size and orientation as the data array.

```
>> T = [10 15 18 22 13]       % array of temperatures
>> Td = [9 11 8 16 3]         % array of dew point

>> m = T > 20                 % output, m, is same size as T
                              % m is a logical variable
m =
      0     0     0     1     0

>> T_subset = T(m)            % subset of T

T_subset =

      22

>> n = Td < 10                % output, n, is same size as Td
                              % n is a logical variable
n =
     1     0     1     0     1

>> Td_subset = Td(n)          % subset of Td

Td_subset =
     9     8     3
```

- **Example 2: Compare two arrays.** Use ***logical operators***: AND, OR, NOT.

```
>> p = (T > 20 | Td < 10)     % Compound expression using OR

p =
     1     0     1     1     1

>> q = m | n                  % same as (T > 20 | Td < 10)

q =
     1     0     1     1     1
```

- **Example 3: Search matrices.** Output of the logical expression is a matrix with the same size as the data matrix. The subset of the data matrix is a column vector.

```
>> r = (A <= 200 | A > 750)          % compound expression

r =

     1     1     0
     0     0     0
     0     1     1

>> subset_A = A(r)                   % subset of A

subset_A =

   100
   200
   800
   900
```

H. SEARCHING DATA ARRAYS

- The three methods for searching a vector or matrix provide ***identical results***.
    1. Using a "for" loop and "if" statement to search a *vector*.
    2. Using a "find" command to search a *vector* or *matrix*.
    3. Using relational and logical operators to search a *vector* or *matrix*.
- There is a special case – searching for NaN – that can only be accomplished with Method 3, using relational and logical operators.

Method 1: Using a "for" loop and "if" statement to search a *vector*.

- This method works on a row or column vector.
- The search criteria may be a compound expression.
- Searching a matrix is much easier to accomplish using the "find" command, so an example using "for' and "if" construction is not provided.

- **Example 1:** Identify the locations in ***vector*** T where T(i) < 0.

```
T = -25:5:100                % Define Temperature Array, T
count = 0                    % Initialize a counter

for I = 1:length(T)          % search each element of T
  if T(I) < 0                % search criteria
    count = count + 1;       % increment counter
    A(count) = I;            % I is the index of T
                             % which meets the criteria
  end
end

subset_T = T(A)              % subset of T
```

- **Example 2:** Identify the locations in ***vector*** T where T(i) < 0 by creating a logical variable.

```
for I = 1:length(T)
   if T(I) < 0
      G(I) = 1;          % G is NOT a logical variable
   else
      G(I) = 0;          % G is NOT a logical variable
   end
end

new_G = logical(G)       % Make new_G a logical variable

subset_T = T(new_G)      % subset of T
```

Method 2: Using a "find" command to search a *vector* or *matrix*.

- **Example 1:** Identify the locations in ***vector*** T where T(i) < 0.

```
T = -25:5:100                % Define Temperature Array, T

A = find(T<0)                % A is the vector of indices
                             % that meet the search criteria

subset_T = T(A)              % subset of T
```

- **Example 2:** Identify the locations in ***matrix*** T where T(i) < 0.

```
T = [-25:5:25; 30:5:80];     % Define Temperature Matrix, T

A = find(T<0)                % A is the vector of indices
                             % that meet the search criteria

subset_T = T(A)              % subset of T
```

- **Example 3:** The search can be accomplished in one step. The output of the find command is the input vector for the data array/matrix.

```
T = [-25:5:25; 30:5:80];      % Define Temperature Matrix, T

subset_T = T(find(T<0))     % the vector that find command output
                            % is the input for the data matrix, T
```

- **Example 4:** The search is using a compound expression.

```
T = [-25:5:25; 30:5:80];      % Define Temperature Matrix, T

A = find(-15 <= T & T > 35)   % compound expression

subset_T = T(A)               % subset of T
```

Method 3: Using relational and logical operators to search a *vector* or *matrix*.

This method is the same for searching vectors or matrices.

- **Example 1:** Searching a vector.

```
T = -25:5:100           % Define Temperature Array, T

L = T < 0               % L is a logical variable
                        % The size and orientation of L
                        % matches the size & orientation of T

subset_T = T(L)         % subset of T; the orientation of the
                        % output matches the orientation of T
```

- **Example 2:** Searching a matrix.

```
T = [-25:5:25; 30:5:80];  % Define Temperature Matrix, T

L = T < 0               % L is a logical variable
                        % The size of matrix L is the size of T

subset_T = T(L)         % subset of T
                        % output is a column vector
```

Special Case: Searching for NaN in a vector or matrix.

- Using the "find" command for NaN does not work.
    - `find(T == NaN)` *does not work*
- There is a built-in function, ***isnan***, to find the NaN's in a vector or matrix.
    - The *output* of ***isnan*** is a ***logical variable***.
- Typically, a programmer finds the NaNs in the dataset for the purpose of making a subset with the NaNs ***excluded***.
    - The ***complement*** operator (~) reverses ones and zeros (1 changed to 0, etc.).
    - The subset with NaN excluded uses the complement of the logical variable generated by ***isnan***.

- **Example 1:** Searching a vector for NaN. The same procedure works for a matrix.

```
T = [0 –5 –10 NaN –6 2]       % Define Temperature Array, T

A = isnan(T)                  % Finds only the NaN

subset_T = T(~A)              % subset excludes NaN

subset_T =
   0   -5   -10   -6    2
```

- **Example 2:** Searching a vector to ***exclude*** NaN. The same procedure works for a matrix.

```
T = [0 –5 –10 NaN –6 2]       % Define Temperature Array, T

A = ~isnan(T)                 % Complement of isnan finds
                              % values not equal to NaN

subset_T = T(A)               % subset excludes NaN
```

- **Example 3:** Searching a vector to ***exclude*** NaN in ***one step***.

```
T = [0 –5 –10 NaN –6 2]       % Define Temperature Array, T

subset_T = T(~isnan(T))       % subset excludes NaN
```

XVIII. DATA WITH MISSING/BAD VALUES AND "NaN"

A. MISSING/BAD DATA AND QUALITY CONTROL

Introduction

Meteorological and oceanographic measured data can have errors in it. There are two types of error:

- Missing values -- sensor is inoperable, a missing data value of -9999 or +999, etc, is inserted in the data set.

- Data value is wrong -- unrealistic values. These are harder to identify. At FNMOC and other data centers, "quality control" routines are used to identify bad data and then fix or eliminate the erroneous values. For data used in course work and theses, you can write MATLAB code to do some primitive quality control checking.

Missing Values

Finding and replacing missing values with the MATLAB missing value, NaN (called Not-a-Number), is a multi-step process.

1. Use the **find** *function* to search the data array or matrix to identify the location in the array/matrix of the -9999, 999, or other "bad" values. The output of the find function is an *array of* ***indices*** which indicate the location within the array/matrix of the "bad" values.

2. Make an array the length of the indices array which have NaN as each element.

3. Using the NaN array, substitute one NaN into each location of "bad" data. Generally, you give this array/matrix of good data and NaNs a new variable name (you never discard the original data).

4. When you plot the arrays with good data and NaNs, the NaNs are not plotted.

5. If you need a plot where the data points are connected by a continuous line, the omitted NaNs cause breaks in the plotted line. There are specific instructions for fixing this plotting problem.

Quality Control

- Prior to computing an equation, use an IF statement to check for the data value to be within certain bounds. For example, check relative humidity values to be within 0-100 percent:

```
%  RH = an array of Relative Humidity values (units of percent)

for  k = 1:length(RH)
   if  RH(k) >= 0 & RH(k) <= 100
      Q(k) = some equation;    % compute using good RH value
   else
      Q(k) = NaN;              % put NaN in array
   end
end
```

- Generally, you need to put NaN into an array to hold the place of a bad value.

    - If you have several data vectors that must be the same length, you must use NaN or the vector will not have the same length as its related data.

    - For example, assume you have RH and Height and other vectors of the same length. If the RH has 5 bad values and you didn't use NaN in the new Q array, then the Q vector will be shorter than the RH and Height, so you couldn't plot Q and Height together.

B. METHOD TO REPLACE MISSING/BAD DATA VALUES

The values of the matrix, **data**, are listed below. The matrix contains the value -9999.0 to indicate missing/bad values. The data type of each column of the matrix is indicated.

```
   Pressure   Height Temperature Dew Point    RH   Mix Ratio   Wind Dir

data =
   1.0e+03 *
    1.0100    0.1540    0.0188    0.0149    0.0781    0.0137    0.2600
    1.0050    0.1250    0.0174    0.0127    0.0739    0.0126    0.2720
    1.0000    0.0950    0.0168    0.0130    0.0783    0.0122   -9.9990
    0.9756    0.3050   -9.9990   -9.9990   -9.9990    0.0107    0.2750
    0.9730    0.3280    0.0142    0.0131    0.0931    0.0106    0.2780
    0.9680    0.3710    0.0138    0.0127    0.0931   -9.9990    0.2830
    0.9670    0.3800    0.0138    0.0117    0.0871    0.0103    0.2840
    0.9630    0.4150    0.0174    0.0014    0.0340    0.0132    0.2880
    0.9610    0.4330   -9.9990    0.0010    0.0281    0.0156    0.2900
    0.9570    0.4690    0.0206   -0.0074    0.0145    0.0162    0.2940
    0.9500    0.5320    0.0240   -0.0180    0.0050    0.0203    0.3010
    0.9420    0.6060    0.0246   -9.9990    0.0097    0.0212   -9.9990
    0.9416    0.6100    0.0245   -0.0083    0.0107    0.0211    0.3100
    0.9380    0.6430    0.0240    0.0010    0.0220   -9.9990    0.3080
    0.9300    0.7180    0.0238   -0.0042    0.0152    0.0204   -9.9990
    0.9250    0.7650    0.0238    0.0018    0.0236    0.0206    0.3000
    0.9092    0.9140    0.0230    0.0038    0.0285    0.0199    0.3050
    0.8778    1.2190    0.0212    0.0080   -9.9990    0.0184    0.2750
    0.8560    1.4370    0.0200    0.0110    0.0561    0.0175    0.2590
    0.8500    1.4980   -9.9990    0.0096    0.0524    0.0172    0.2550
```

Procedure to Replace Bad/Missing Data (-9999.0) with NaN

- Replace all –9999 with NaN before breaking the matrix into column vectors.
    - Use the ***find*** function to find the *indices* where –9999 values exist. Store these indices in a variable (index_bad_values).

      ```
      index_bad_values = find(data == -9999);
      ```

    - Use the ***size*** function to find the size of the variable, index_bad_values. The output of the size function are the number of rows and columns.

      ```
      [row,col]=size(index_bad_values);
      ```

- Replace the bad values of the data matrix with NaN. The index_bad_values variable is used to indicate which locations in the data matrix must be replaced. NaN*ones(row,col) makes a vector of NaN which has the exact number of NaN needed. The function ***ones*** is used to create the vector and NaN multiplies each element.

```
data(index_bad_values) = NaN*ones(row,col);
```

- Example code:

```
index_bad_values = find(data == -9999);
[row,col]=size(index_bad_values);
data(index_bad_values) = NaN*ones(row,col);
```

- If the matrix is broken into column vectors before –9999 are replaced, then ***each*** column vector must be searched and bad values replaced. Replace all –9999 with NaN before breaking the matrix into column vectors. The location of bad values in each column vector ***may be different***.

- Example code: pres and tmpc are column vectors.

```
index_bad_p =  find(pres == -9999);
[row,col]=size(index_bad_p);
pres(index_bad_p) = NaN*ones(row,col);

index_bad_t =  find(tmpc == -9999);
[row,col]=size(index_bad_t);
tmpc(index_bad_t) = NaN*ones(row,col);
```

C. EXCLUDING "NANS" IN PLOTS

Introduction

- Plotting vectors that include NaN may cause breaks in the plot. To fix this, you must create new vectors that only have good values (no NaN).

- For example, if we are plotting column vectors of temperature (T) and Dew Point (TD) versus height (H).

- There is a one-to-one correspondence between the values in each T and TD vector and height.

- Since you can only plot vectors of equal length, you must also exclude the corresponding height values for each bad T and TD value.

- We must use the ***isnan*** function to locate NaNs in each vector. The ***find*** function does not work with NaNs. The output of the ***isnan*** function is a logical vector, of the same size as the data vector, where 1 indicates NaN and 0 indicates other than NaN.

Steps:

1. Locate the NaNs in the temperature vector: **`T_nan = isnan(T)`**.

2. Make a new temperature vector which do not include NaN. Use the complement (reverse 1 and 0) of the T_nan vector: **`new_T = T(~T_nan)`**

3. Make a new height vector that *excludes* the *corresponding* elements as the temperature vector. The new height vector is the same length as the *new* temperature vector: **`new_H1 = H(~T_nan)`**

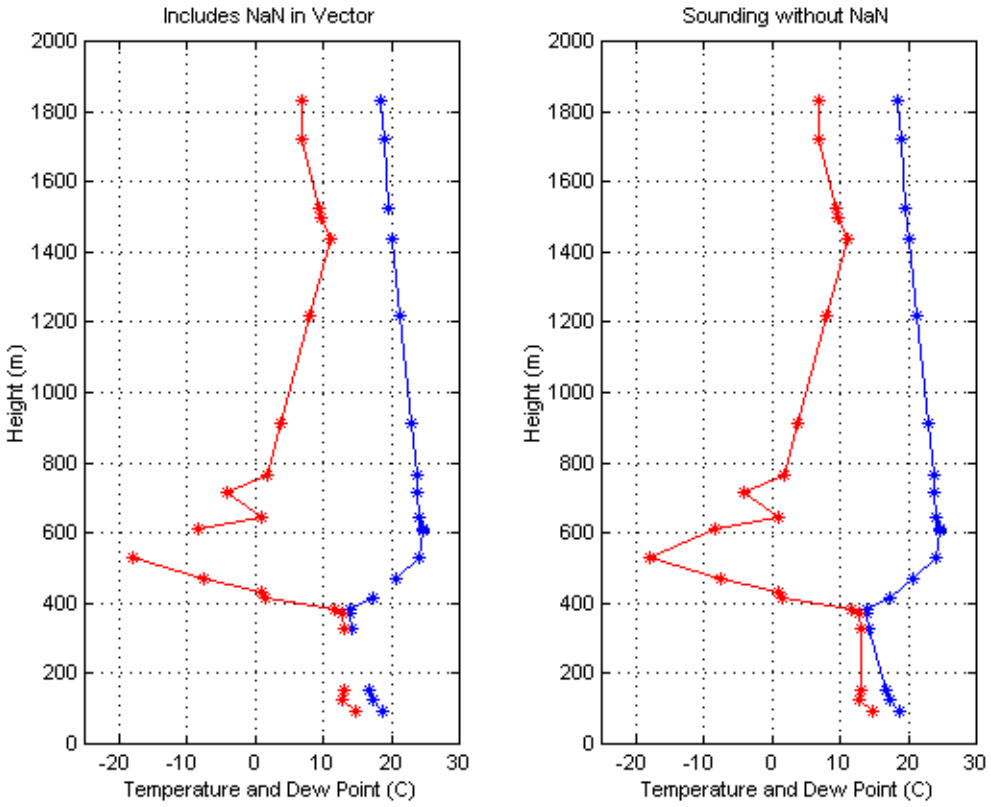
4. Do the same procedure for the dew point vector. Since there are a different number of NaNs in this vector than temperature, we need a different new height vector to correspond to the new dew point vector. Here are the commands:

```
Td_nan = isnan(TD)
new_TD = TD(~Td_nan)
new_H2 = H(~Td_nan)
```

5. Plot the new temperature and dew point vectors versus their *corresponding* height vectors.

```
plot(new_T, new_H1,'b*, new_TD , new_H2,'r-d')
```

Figure 11. Example of plotted vectors with and without "NaNs". The left plot has breaks in the lines where "NaNs" occur in the data vectors. The right plot has the "NaNs" removed from each vector and a straight line is plotted between existing data points.



XIX. PROGRAM DESIGN EXAMPLE

Problem: Plot rawinsonde data.

Student questions:

1. Where is the rawinsonde data file?
2. What form is the rawinsonde data in?
3. Should the program be designed to plot more than one rawinsonde?
4. What data variables do you want plotted?
5. What are the requirements for layout of the plot(s)?
6. Need a hardcopy or digital copy of the plot(s)?
7. Should the MATLAB workspace be saved for future use?

Steps of the Program (pseudocode):

1. Ingest the rawinsonde data -- take from UNIX and put into the MATLAB workspace.
2. In MATLAB, make column vectors of the rawinsonde measurements.
3. Compute additional variables from the rawinsonde data.
4. Save the workspace as a .mat file.
5. Make the plot(s).
6. Print the plot(s) or save the digital copy of the plot(s).

Each Step is Designed in Greater Detail (pseudocode):

1. Data Ingest:

    - If rawinsonde data is already in a MATLAB ".mat" file, then load it with the "load" command.
    - If rawinsonde data is in an ASCII text file with (1) data in columns (2) contains only numbers, no letters or slashes, etc, and (3) no text header lines, then load the file into MATLAB with the "load" command.
    - If rawinsonde data is in an ASCII text file with (1) data in columns (2) contains only numbers, no letters or slashes, etc, and (3) contains text header lines, then edit the ASCII file and put % signs in Column 1 of each text header line (making the header lines appear as comments to MATLAB). Save the new data file and load the new file into MATLAB with the "load" command.
    - If the rawinsonde data cannot be easily changed into a format suitable for the "load" command, then a lengthy data "decoder" program (M-file) must be written.

2. Once the data is loaded into MATLAB, break the matrix into column vectors using the "colon method".

3. Write code to compute additional meteorological variables using the column vectors already in the MATLAB workspace. If necessary, create character string variables which contain the rawinsonde location and date/time of the sounding (will be needed for the plot).

4. Decide on a file name and save the workspace using the "save" command.

5. Design the plot(s).

    - One plot per page?
    - Multiple plots? Use subplot command. Design each plot on the page. What is plotted on the X-axis, what is the Y-axis variable?
    - What X-axis, Y-axis label names are needed? What plot title is needed?
    - Is a legend needed?

6. Print or save the plot digitally? Print in color? Which digital format to save the plot?

Write and Test Each Task:

1. Take the pseudocode describing each task and write the MATLAB code.
2. Test each section of code independently.
3. Combine all tasks into the final program.

XX. DEBUGGING SUGGESTIONS

1. Clear your workspace and re-execute your M-file. See if the problem is reproduced. While debugging, a programmer may change variables in the MATLAB workspace that causes problems for other lines of M-file.

2. **Do not** tear apart your original M-file. Copy the original program and make changes to a *test* M-file. Too often, students destroy their original code when the error is minor, but undetected for several minutes.

3. Take a break. Come back refreshed and *relaxed*.

4. Make a plan for testing your code. Write down the results of each test. This makes your testing logical and thorough.

5. Test small portions of the code in your M-file.

6. Print out the most recent version of your M-file. Make sure your loops and if blocks start and end properly.

7. Once you've isolated the section of the code that seems to be the problem, double-check your logic.

    - Do you understand the logic of the homework or lab assignment? Be sure you understand the logic in math/physics terms before programming.
    - Is each line of your code performing the designed math/physics requirement?

8. Clear your workspace and rerun the corrected portions of your test M-file. Save your workspace as a ".mat" file (so someone else can later review your test results).

9. Have someone else look at your code. Explain your logic and how you've tested it.

XXI. FORMATTED OUTPUT

A. INTRODUCTION

- Programmers write to the ***screen*** with the ***disp*** and ***sprintf*** functions.
    - The ***disp*** function (course notes on p. 38).
    - The ***sprintf*** function is more versatile, but is more difficult to use.
- Programmers write to an *open **ASCII text file*** with the ***fprintf*** function.
    - To open an ASCII file, use the ***fopen*** function (notes, p. 131).
    - To close an ASCII file, use the ***fclose*** function (notes, p. 132).
    - On-line help provides additional details of ***fopen*** and ***fclose***.
- The ***sprintf*** and ***fprintf*** functions require a format string to specify the precise number of digits and spacing to display numbers.

B. "sprintf" FUNCTION

- The ***sprintf*** function requires a *format string* to specify the precise format to print the value of each variable.
    - Syntax: **`sprintf(`*`format`*`,A)`**
    - The variable, A, is printed using the format prescribed in the format string.
    - The *format* is a *character string*, enclosed by single quotes. Format strings are described in the next section.
    - The *output* of the function *sprintf* is a character string, which can be assigned to a variable. A semicolon (;) will suppress the output (not write to the screen).
    - If an output variable is not assigned, MATLAB assigns variable "ans" and the string is written as "ans ="

- **Example 1: Printing scalar variables.**

  Using variables *pres* and *temp*, print pressure using 7 characters and temperature with 6 characters:

```
sprintf('Pressure = %7.2f  Temperature = %6.2f',pres,temp)

ans =
Pressure =  519.65  Temperature = -10.37
```

- **Example 2: Assign the output string to a variable.**

  Assign the output string to a variable, use a semicolon to suppress the screen print, and use disp to print to the screen. This method avoids extra lines with "ans= "

```
S = sprintf('Pressure = %7.2f  Temperature = %6.2f',pres,temp);
disp(S)

Pressure =  519.65  Temperature = -10.37
```

- **Example 3: Writing arrays *with* a "for" loop.**

  Arrays must have equal length. A header for the output table is written to variable string S1.

```
S1=sprintf('Pres     Temp');    % create a header
disp(S1)                        % print the header

for i=1:length(pres)
    S2=sprintf('%7.2f %6.2f', pres(i),tmpc(i));
    disp(S2)
end


Pres     Temp
1011.90  19.20
1000.40  16.20
 997.50  15.90
 900.30  13.30
 601.10  -0.10
 572.80  -1.30
 400.20 -12.30
```

- **Example 4: Writing matrices *without* a "for" loop.**

  sprintf can print *vectorized* (without a for loop) to write a *matrix*. The format string is cycled through the elements of A, columnwise, until all the elements are used.

  - If *matrix* A has 10 rows and 2 columns, the 10 elements in column 1 are printed before the elements in column 2 are printed.
  - To make a table with 2 ***columns***, you need to create *matrix* A with 2 ***rows***. Each column of A will have only 2 values. When *sprintf* prints each column of A it is written as one row of a table.

- Using **Row Vectors** pres and temp, create A with 2 rows:
  row 1 = pres, and row 2 = temp.

```
P = pres';        % Make column vector a row vector
T = temp';        % Make column vector a row vector
A = [P; T];       % Matrix A has 2 rows

S=sprintf('%7.2f %6.2f\n', A);
disp(S)

1011.90  19.20
1000.40  16.20
 997.50  15.90
 900.30  13.30
 601.10  -0.10
 572.80  -1.30
 400.20 -12.30
```

- **Example 5: Printing character string variables.**

  Use the format **`%s`** to print character string variables.

```
filename='oakland_snd.data';     % character string variable
datetime='31/1200 UTC Oct 96';   % character string variable

S = sprintf('Decoding file: %s, for %s',filename,datetime);
disp(S)

Decoding file: oakland_snd.data, for 31/1200 UTC Oct 96
```

C. FORMAT STRINGS

- The functions **sprintf, fprintf**, and **fscanf** each need a **format** string.

```
sprintf(format,A)
fprintf(fid,format,A)
fscanf(fid,format,A)
```

- ***format*** is a string containing ordinary characters and/or conversion specifications. Ordinary characters include the normal alphanumeric characters, and escape characters.
- Escape characters include:

```
\n   new line
\t   horizontal tab
\b   backspace
\r   carriage return
\f   form feed
\    backslash
```

- Conversion specifications involve the character **%**, *optional width field*, and conversion characters. Legal *conversion characters* are:

```
%e    exponential notation
%f    fixed point notation
%g    %e or %f, whichever is shorter;
      (insignificant zeros do not print)
%s    character strings
```

- Between the **%** and the conversion character (e, f, or g), you can add one or more of the following characters:
    - Digit string to specify a *field width*, which includes the decimal point and +/- sign.
    - *Period* (.) to separate the field width from the next digit string.
    - Digit string specifying the precision (i.e., *the number of digits to the right of the decimal point*). Decimals are rounded to print out the correct number of digits.
    - Minus sign (-) to specify left adjustment of the converted argument in its field.

- In the format statement, you must specify a sufficient number of characters to allow the largest number in the data set to be printed.

  `%7.2f` indicates a floating-point number with:

  - 7 spaces allocated to the number (including the decimal point, +/- sign).
  - 2 digits to the right of the decimal point.
  - The negative number -234.67 fits the `%7.2f` format, but -2345.78 needs 8 characters to print and only 7 are allocated.
  - The positive number 1234.67 fits the `%7.2f` format, but 12345.78 needs 8 characters to print and only 7 are allocated.
  - The number 12.01 fits the `%7.2f` format, and two blank spaces are padded to the left of the number. If the format was **`%-7.2f`**, then the number prints in the first allocated character location and tow blank spaces are added at the end (right) of the number.

- Spaces included in the format string will be included in the written output. To add 3 spaces between output variables, add 3 spaces in the format string. The tab option, **`/t`**, is a convenient way to space columns of output.
- To print a number as an integer, specify zero digits after the decimal point in a fixed point notation (`%5.0f`).

- **Example 1: Printing in decimal "f" format.** Using variables pres, temp, print pressure using 7 characters and temperature with 6 characters:

```
sprintf('Pressure = %7.2f  Temperature = %6.2f',pres,temp)

ans =
Pressure =  519.65  Temperature = -10.37
```

- **Example 2: Printing in exponential "e" format.**
  - The "e" format has **eight** digits of "overhead". To print a number with 4 digits after the decimal point, the format field width must be 12 (4 decimal digits + 8 overhead digits).
  - The overhead includes +/- sign at the beginning of the number, the first digit, the decimal point, and room for the exponent (e+123 or e-123).

```
sprintf('Temperature =%12.4e',temp)

ans =
Temperature =-6.5250e+001
```

- **Example 3: Print as an integer**. Specify zero digits after the decimal point in a fixed point notation (`%5.0f`). A real number will be rounded to print as an integer.

```
sprintf('%7.2f %6.2f %4.0f', pres,tmpc,relh)

ans =
  17.70 -57.70   36
```

- **Example 4: Print multiple variables.** To print multiple variables with the same format width and precision, you must explicitly type each format option. There is no shortcut, as in other languages. This example uses the tab option.

```
sprintf('%12.4e\t %12.4e\t %12.4e\t %12.4e',T1, T2, T3, T4)

ans =
-1.6500e+001   -4.0010e+001   2.4750e+001   2.5006e+001
```

- **Example 5: Reusing the format string.** When printing with a "for loop", the format string is reused many times. Arrays must have equal length. A header for the output table is written to variable string S1.

```
S1=sprintf('Pres     Temp');    % create a header
disp(S1)                        % print the header

for i=1:length(pres)
    S2=sprintf('%7.2f %6.2f', pres(i),tmpc(i));
    disp(S2)
end

Pres     Temp
1011.90  19.20
1000.40  16.20
 997.50  15.90
 900.30  13.30
 601.10  -0.10
 572.80  -1.30
 400.20 -12.30
```

- **Example 6: Printing character string variables.** Use the format **`%s`** for character string variables.

```
filename='oakland_snd.data';     % character string variable
datetime='31/1200 UTC Oct 96';   % character string variable

S = sprintf('Decoding file: %s, for %s',filename,datetime);
disp(S)

Decoding file: oakland_snd.data, for 31/1200 UTC Oct 96
```

- **Example 7: Start a new line of output.** Use the \n option:

```
S = sprintf('Decoding file: %s \n for %s',filename,datetime);
disp(S)

Decoding file: oakland_snd.data
 for 31/1200 UTC Oct 96
```

- **For more information** about format strings, refer to the printf and fprintf routines in a reference manual for the C language.

D. "fprintf" FUNCTION

- Write to an *open* file with **fprintf**.

  `fprintf(fid,`*`format`*`,A)`

- The only difference between *fprintf* and *sprintf* is the file identifier, *fid*, is used.
  - The file identifier, *fid*, is a number, specified by MATLAB as output of the ***fopen*** function, which associates the file name and pointer for the file.
  - On-line help provides details of ***fopen*** and ***fclose***.

- **Example 1: Writing *with* a "for" loop.**

  Using arrays *pres* and *temp*, write one header line and all data values.

```
outfile = 'sample.out';          % define file name
fid = fopen(outfile,'w');        % open the ASCII file

fprintf(fid,'Pres     Temp \n'); % print header line

for i=1:length(pres)              % loop to print data
   fprintf(fid,'%7.2f %6.2f \n', pres(i),tmpc(i));
end

fclose(fid);                      % close ASCII file
```

- **Example 2: Writing *without* a "for" loop.**
  - **fprintf** can write *vectorized* (without a for loop).
  - The format string is recycled through the elements of matrix A (**columnwise**) until all the elements are used.
  - If matrix A has 10 rows and 2 columns, the 10 elements in column 1 are printed before the elements in column 2 are printed.
  - To make a table with 2 **columns**, you need to create matrix A with 2 **rows**. Each column of A will have only 2 values.

Using **Row Vectors** pres and temp, create A with 2 rows:
row 1 = pres and row 2 = temp.

```
P = pres';       % Make column vector a row vector
T = temp';       % Make column vector a row vector
A = [P; T];      % A has 2 rows

outfile = 'sample.out';      % define file name
fid = fopen(outfile,'w');    % open ASCII file

fprintf(fid,'%7.2f %6.2f \n', A);  %print all values
                                   % in matrix A
fclose(fid);                 % close ASCII file
```

- **Example 3: Writing character strings to a file.**

```
filename='oakland_snd.data';
datetime='31/1200 UTC Oct 96';
fprintf(fid,'Decoding file: %s, for %s',filename,datetime);
```

XXII. READING ASCII TEXT DATA FILES

A. INTRODUCTION

Many types of meteorological and oceanographic real-time observations, such as rawinsondes, expendable bathythermograph (XBT) and sensors on ships and buoys, store the observations in an ASCII text file.

- Each observation has an associated date and time, location (latitude and longitude) and elevation above the earth's surface or depth below the ocean surface.

There are many formats for the ASCII files, but typically data will be listed in columns. For simple formats, the "load" function may be used. Other formats require a more sophisticated program to decode the data file.

- Use the "load" function for:
    - Data in columns, without (1) text header lines, and (2) numbers (-99, 99, etc.) to indicate missing values.
    - Data in columns, with the percent sign (%) before each text header line, and numbers (-99, 99, etc.) to indicate missing values.
    - If the text characters, such as ///// for missing values, can be replaced by 99999 or -9999, use a text editor to make the changes. Then use the "load" function.
- A more sophisticated program is needed to decode file formats where:
    - Information in the text header lines, such as station identifier, latitude, longitude, elevation, date and time, must be decoded.
    - Text characters, such as ///// to indicate missing values or letters to indicate quality control flags, occur in the columns of data.
- When a sophisticated decoder program is needed, the data file must be opened and read using functions described in this chapter.
- Once a data file is open, each line of alpha-numeric text is read with the "**fgetl**" function and lines of numbers in columns are read with the "**fscanf**" function.

B. OPEN FILE WITH "fopen" FUNCTION

The **"fopen" function** opens the designated file and automatically assigns a number, called a *file identifier*, which is the pointer MATLAB uses to refer to the file instead of using the actual file name.

```
fid = fopen('sample_data.txt','r')
```

- The file identifier number is an output of the *fopen* function and a variable name is specified when the function is called. A typical variable name for the file identifier is "*fid*".
- The file name is specified in single quotes.
- The file permission is the second function input, and is also specified in single quotes. The most common permissions are listed below. See on-line help for the complete list of permissions.

```
'r'     read
'w'     write (create if necessary)
'a'     append (create if necessary)
        append is permission to write at the
        END of the existing file
```

- The options listed above do not allow reading and writing to the same file at the same time.
- Typically, a user will read from a data file and write or append to a separate output file.
- If the file name is specified in a variable, then the variable is used in the fopen function.
- If two files are open at one time, such as one input and one output file, use different variable names for the two file identifiers.

```
inputfile  = 'input_data.txt'
outputfile = 'output.txt'

fid_in  = fopen(inputfile,'r')
fid_out = fopen(outputfile,'w')
```

- If there is a problem opening a file, the number –1 is assigned to the file identifier.

C. CLOSE FILE WITH "fclose" FUNCTION

The **"fclose" function** closes the file designated by the file identifier variable. There is an integer generated as output of the function -- zero, 0, indicates the file closed correctly, -1 indicates a problem closing the file.

```
fclose(fid)
```

- Each file that is open must be closed by a separate call to the *fclose* function.

```
fclose(fid_in)
fclose(fid_out)
```

- Close all files prior to ending the program.

D. READ LINE OF TEXT WITH "fgetl" FUNCTION

The **"fgetl" function** reads one line of text from an open file and reads ASCII text only. The last character of the function name is a lowercase L.

- The input to the function is the file identifier variable for the data file to be read.
- The function needs an output variable to be designated, and the output variable, a character string, will contain the ASCII text line.

```
line = fgetl(fid);
```

- Text lines must be read ***sequentially*** in an ASCII data file. To read the fourth line, the first three lines must be read.

```
line1 = fgetl(fid);        % read 1st line
line2 = fgetl(fid);        % read 2nd line
line3 = fgetl(fid);        % read 3rd line
line4 = fgetl(fid);        % read 4th line
```

- The output variable is a character string and substring elements may be extracted using the rules for character strings on p. 94. Examples are listed on p. .
- The output variable name can be the same (instead of line1, line2) is the data is extracted from the variable character string before the next line of data is read.

E. READ DATA COLUMNS WITH "fscanf" FUNCTION

The **"fscanf" function** is used for reading **columns of data** (all numbers) from an input data file. There can be no text characters in the rows and columns of data to be read with "fscanf".

- If the data file contains text header lines, read each header line with the "**fgetl**" function. When the next data line to be read starts the rows and columns of numerical data, issue the "**fscanf**" command.
- Function input:
    - File identifier variable for an *open* data file.
    - Format string, as explained on p. 124, and in the following examples.
    - Size of the output matrix, based on the number of rows and columns of data.
        - Instead of counting the number of input data rows, the number infinity, "inf", may be used, and MATLAB will read until the end of the data file is found.
- Function output:
    - Variable for the matrix of data, containing each ***row*** of the data file stored as a ***column*** of the output matrix (row 1 of input file is stored in column 1 of output matrix).
    - The output matrix must be ***transposed*** to re-orient the matrix data to look like the input data file (column 1 of input file in column 1 of output matrix).
- Read data with a function call similar to:

```
A = fscanf(fid,'format',[m,n])
```

- [m,n] defines the size of the output matrix, A: ***m rows*** and ***n columns***.
- The size of output matrix is determined from the input data file.
    - The input data file, which has ***m columns*** and ***n rows***.
    - If the input file has 6 *columns* and 500 *rows*, m = 6, n = 500.
    - If you are not sure how many rows of data are in the input file, define n to be infinite, i.e. [m,inf].
    - The number of rows, n, can be infinite, but not the number of columns, m.
    - The output matrix must be transposed, so the matrix has ***m columns*** and ***n rows*** to match the input data file.

- The format statement does not need to specify the width and number of digits after the decimal point, but it must specify the *type of number* being read in *each column* of the input data file.
    - For five columns of decimal data, use the format string:

```
'%f %f %f %f %f'
```

- Legal *conversion characters* are:

```
%e    exponential notation
%f    fixed point notation
%g    %e or %f, whichever is shorter;
      (insignificant zeros do not print)
```

Example 1: File with No Text Header Lines

- Assume the input file has 6 columns and 500 rows. Data is in either floating point (decimal) or exponential format, so we can use the **g** format, which accepts either **f** or **e** format.

```
infile = 'sample.data'
fid = fopen(infile,'r');

A = fscanf(fid,'%g %g %g %g %g %g \n',[6,500]);
A = A';                                  % Transpose A

fclose(fid);
```

- Note: to make this code more general (not limited to files <= 500 lines), use the n = infinite option.

```
A = fscanf(fid,'%g %g %g %g %g %g \n',[6,inf]);
```

Example 2: File with Header Lines

Assume the input file has 4 header text lines followed by 6 columns and 500 rows of data.

```
infile = 'sample.data'
fid = fopen(infile,'r');

line=fgetl(fid);       % read header line #1
line=fgetl(fid);       % read header line #2
line=fgetl(fid);       % read header line #3
line=fgetl(fid);       % read header line #4

A = fscanf(fid,'%g %g %g %g %g %g \n',[6,inf]);
A = A';
fclose(fid);
```

XXIII. EXAMPLES OF DECODERS FOR ASCII DATA FILES

A. INTRODUCTION

The following four examples are all rawinsondes data, but the techniques applied in these examples can be used for most types of ASCII text data files.

- All four examples have text header lines preceding the columns of data.
    - Each text header line is read with the "fgetl" function.
    - Station identifier, date/time, and other information are extracted from the header using character string manipulation techniques.
    - Latitude, longitude and elevation values are converted from character strings to real numbers using the "str2num" function.
    - All header lines must be read with "fgetl" before the columns of data can be read.
- In the first two examples, there are no text characters, such as '///', used to denote missing values.
    - Since only numbers exist in each column, the "fscanf" function is used read the columns of data.
    - Missing values are denoted with –9999.99 or 99999. Missing values must be set to NaN using methods described on p. 114.
- In examples 3 and 4, the text characters '////' are used to denote missing values in the columns of data.
    - The "fscanf" function cannot be used to read '///'.
    - Each line of data must be read as a text string, using "fgetl", and the data values extracted with string manipulation.
    - Each data string must be checked for '///' before using the "str2num" function. If the string contains '///' and not a data value, NaN is stored in the data array.

B. EXAMPLE 1: MISSING DATA DENOTED BY –9999.99

File = sample_data1.txt

```
SNPARM = PRES;HGHT;TMPC;DWPC;RELH;DRCT;SPED
 STID = OAK           STNM =    72493   TIME = 990126/0000
 SLAT =  37.73     SLON =  -122.22   SELV =     3.0
 STIM =     0
      PRES     HGHT     TMPC     DWPC     RELH     DRCT     SPED
   1012.00     3.00    11.60     5.60    66.61   180.00     5.00
   1000.00   104.00    10.00     3.00    61.76 -9999.99 -9999.99
    925.00   745.00     4.20     3.20    93.20   225.00     7.00
```

Decode with: sample_decoder1.m

```
% sample_decoder1.m   M. Jordan  2/22/2002
% Decodes data in the exact format specified in file=sample_data1.txt.

infile='sample_data1.txt';        % Define ASCII data file name
fid = fopen(infile,'r');          % Open ASCII file with read option

line1 = fgetl(fid);              % Reads 1st line as a character string
line2 = fgetl(fid);              % Reads 2nd line as a character string
line3 = fgetl(fid);              % Reads 3rd line as a character string
line4 = fgetl(fid);              % Reads 4th line as a character string
line5 = fgetl(fid);              % Reads 5th line as a character string

stid = line2(9:11);           % Station ID is a character string
stnm = line2(33:37);          % Station Number is a character string
datetime = line2(48:58);      % Date/Time of data is a character string
lat = line3(9:14);            % Latitude is a character string
lon = line3(28:34);           % Longitude is a character string
elev = line3(45:51);          % Elevation (m) is a character string

%  read 7 columns of data, stores in matrix A
%     [7,inf] allows infinite number of lines
%     data columns in the ASCII file are stored as ROWS in matrix A

A = fscanf(fid,'%f %f %f %f %f %f %f \n',[7,inf]);

%  transpose matrix A, so the columns of data in the ASCII file
%  are columns in matrix A

data = A';

fclose(fid);                      % close ASCII file

%  break matrix A into column vectors
pres=data(:,1);           % Pressure (mb)
hght=data(:,2);           % Height (meters)
tmpc=data(:,3);           % Temperature (C)
dwpc=data(:,4);           % Dew Pt (C)
relh=data(:,5);           % Relative Humidity (%)
drct=data(:,6);           % Wind Direction (degrees)
sped=data(:,7);           % Wind Speed (m/s)
```

C. EXAMPLE 2: MISSING DATA DENOTED BY 9999

File = sample_data2.txt

```
Started at:      28 AUG 93 23:52 GMT
   Time AscRate Hgt/MSL Pressure   Temp  RH   Dewp  Dir Speed     WndStat
 min  s    ft/s      ft      hPa   degC   %   degC  deg   kts
   0  0     0.0       7   1010.3   21.9  65   15.0  260   8.9 000000000000
   0  2    21.3      49   1008.8   20.7  70   15.1  265   9.3 100100111100
   0  4    15.6      69   9999.9   20.1  74   15.3  999 999.9 100100111100
   0  6    13.7      89   1007.3  999.9 999  999.9  269   9.7 100100111100
   0  8    13.1     112   1006.6   18.9  77   14.8  271   9.9 100100111100
   0 10    12.5   99999   1005.8   18.8  79   15.1  271  10.1 100100111100
   0 12    12.0     151   1005.1   18.7  80   15.2  271  10.3 100100111100
   0 14    12.0     174   1004.4   18.7  81   15.4  271  10.5 100100111100
   0 16    12.1     200   1003.4   18.7  82   15.6  269  10.5 100100111100
   0 18    11.8     220   1002.6   18.6  83   15.7  268  10.5 100100111100
```

Decode with: sample_decoder2.m

```
% sample_decoder2.m   M. Jordan  2/22/2002
% Decodes data in the exact format specified in file=sample_data2.txt.
% Reads data with "fscanf"

infile='sample_data2.txt';       % Define ASCII data file name
fid = fopen(infile,'r');         % Open ASCII file with read option

line1 = fgetl(fid);          % Reads 1st line as a character string
line2 = fgetl(fid);          % Reads 2nd line as a character string
line3 = fgetl(fid);          % Reads 3rd line as a character string
datetime = line1(18:36)      % Date/Time of data is a character string

%  read 11 columns of data, stores in matrix A
%     [11,inf] allows infinite number of lines
%     data columns in the ASCII file are stored as ROWS in matrix A

A = fscanf(fid,'%f %f %f %f %f %f %f %f %f %f %f \n',[11,inf]);

%  transpose matrix A, so the columns of data in the ASCII file
%  are columns in matrix A

data = A';
fclose(fid);              % Close data file

%  break matrix A into column vectors
hght=data(:,4);           % Height (feet)
pres=data(:,5);           % Pressure (mb)
tmpc=data(:,6);           % Temperature (C)
relh=data(:,7);           % Relative Humidity (%)
dwpc=data(:,8);           % Dew Pt (C)
drct=data(:,9);           % Wind Direction (degrees)
sped=data(:,10);          % Wind Speed (knots)
```

D. EXAMPLE 3: MISSING DATA DENOTED BY SLASHES

For data in the format listed below, use the sample program listed below. Missing data is denoted with slashes (////) for each missing character.

File = sample_data3.txt

```
Sounding program REV 7.62 using Omega

Ship     : PT SUR
Location : 33.38 N 177.77 W      3 m

Started at:       8 APR 96 20:57 GMT
   Time AscRate Hgt/MSL Pressure   Temp  RH   Dewp  Dir Speed     WndStat
 min  s     m/s       m       mb   degC   %   degC  deg   m/s

   0  0     0.0       3   1016.8   16.5  72   10.5  276   3.0 -----------
   0  2     2.0       7   1016.2   16.0  61    9.0  ///  //// --CDEFGH---
   0  4     1.5   /////   1016.0   15.9  61    8.7  ///  //// --CDEFGH---
   0  6     2.3      17   //////   15.1  63    8.2  ///  //// --CDEFGH---
   0  8     2.1      20   1014.7   14.7  64  /////  ///  //// --CDEFGH---
   0 10     2.5      28   1013.7  /////  66    8.0  187   3.8 --CDEFGH---
   0 12     2.4      32   1013.2   14.0  68    8.2  ///  //// --CDEFGH---
   0 14     2.2      34   1013.0   13.9 ///    8.4  ///  //// --CDEFGH---
   0 16     2.3      40   1012.2   13.8  71    8.7  ///  //// --CDEFGH---
```

Decode with: sample_decoder3.m

```
%  file = sample_decoder3.m
%  written by:  M. Jordan  02/25/2002
%
%  Purpose: Decode the sounding 'sample_data3.txt',
%           which contains ////'s to indicate missing values.
%
%      1.  Each sounding data line must be read as a character string.
%      2.  Data values must be located within the string and
%          converted from character to number (str2num function).
%      3.  NaN = Missing values
%
%  Input:
%        None.  Filename is "hardwired".
%
%  Output:
%        stid      = station ID character string
%        datetime  = date/time character string
%        latstr    = latitude character string
%        lonstr    = longitude character string
%        elevstr   = station elevation (m) character string
%        iline     = number of sounding lines decoded
%        lat       = latitude - real number
%        lon       = longitude - real number
%        elev      = station elevation - real number
%        hght      = height (m)
%        pres      = pressure (mb)
%        tmpc      = temperature (C)
%        relh      = relative humidity (%)
%        dwpc      = dew point temperature (C)
%        drct      = wind direction (degrees)
%        sped      = wind speed (m/s)
```

```
%           ... open the data file, read only
fid_in = fopen('sample_data3.txt','r');

%           ... read each text header line
line1 = fgetl(fid_in);
line2 = fgetl(fid_in);
line3 = fgetl(fid_in);
line4 = fgetl(fid_in);
line5 = fgetl(fid_in);
line6 = fgetl(fid_in);
line7 = fgetl(fid_in);
line8 = fgetl(fid_in);
line9 = fgetl(fid_in);

%  decode the information in the sounding header
%
stid     = line3(12:17);      % station ID character string
latstr   = line4(12:16);      % latitude character string
lonstr   = line4(20:25);      % longitude character string
elevstr  = line4(29:34);      % station elevation (m) character string
datetime = line6(18:36);      % date/time character string

% convert string lat/lon/elev to real numbers

if (line4(18:18) == 'N')
   lat = str2num(latstr) ;       % N. Hemisphere Latitude is positive
elseif (line4(18:18) == 'S')
   lat = -1.0*str2num(latstr);   % S. Hemisphere Latitude is negative
else
   lat = NaN;                    % Latitude is missing
end


if (line4(27:27) == 'E')
   lon = str2num(lonstr);        % E. Hemisphere Longitude is positive
elseif (line4(27:27) == 'W')
   lon = -1.0*str2num(lonstr);   % W. Hemisphere Longitude is negative
else
   lon = NaN;                    % Longitude is missing
end

elev = str2num(elevstr);         % convert elevation to real number


%           ... initialize a line counter
iline=0;

%
%           ... This WHILE LOOP to READ EACH DATA LINE is an
%            infinite loop, since "while 1" is always true.
%            The way to exit from the loop is with an if, break.
%            The "isstr" function will test if line is a character
%            string.  The if ~isstr(line) logic means if line is
%            NOT (~) a string.  So if ~isstr(line) is TRUE when
%            line is NOT a string, and is the end-of-file.
%            When end of file is reached, "break" exists the WHILE Loop.

while 1

%          ...      Read each line as a character string
   line=fgetl(fid_in);
```

```
%           ... Check for the effective "end-of-file"
   if ~isstr(line), break, end

%           ... Increment line counter.  This is the index for each
%            data array
   iline = iline+1;

%          ... IF logic to check for missing values, ////.  If not
%            '////', then convert string to number.

   if line(19:23) == '/////'                      % height (m)
       hght(iline) = NaN;
   else
       hght(iline) = str2num(line(19:23));
   end


   if line(27:32) == '//////'                     % pressure (mb)
       pres(iline) = NaN;
   else
       pres(iline) = str2num(line(27:32));
   end


   if line(35:39) == '/////'                      % temperature (C)
       tmpc(iline) = NaN;
   else
       tmpc(iline) = str2num(line(35:39));
   end


   if line(41:43) == '///'                        % relative humidity (%)
       relh(iline) = NaN;
   else
       relh(iline) = str2num(line(41:43));
   end


   if line(46:50) == '/////'                      % dew point (C)
       dwpc(iline) = NaN;
   else
       dwpc(iline) = str2num(line(46:50));
   end


   if line(53:55) == '///'                       % wind direction (deg)
       drct(iline) = NaN;
   else
       drct(iline) = str2num(line(53:55));
   end


   if line(58:61) == '////'                      % wind speed (m/s)
       sped(iline) = NaN;
   else
       sped(iline) = str2num(line(58:61));
   end

end          % This is the end of the WHILE Loop

%           ... The IF, BREAK jumps to the command following this line.

fclose(fid_in);       % close input file
```

```
%     transpose row vectors into column vectors

hght = hght';
pres = pres';
tmpc = tmpc';
relh = relh';
dwpc = dwpc';
drct = drct';
sped = sped';

sprintf('Number of lines in the input sounding: %5.0f',iline)

%      clear unnecessary variables
clear fid_in ans iline
clear line line1 line2 line3 line4 line5 line6 line7 line8 line9
% ------------------------------------------------------------------
```

E. EXAMPLE 4: MISSING DATA DENOTED BY SLASHES

For data in the format listed below, use the sample program listed below. Missing data is denoted with slashes (////) for each missing character.

File = sample_data4.txt

```
 STID = NSI        STNM =    72291   TIME = 930824/0000
 SLAT =  33.25     SLON =  -119.45   SELV =   154.0

    HGHT   PRES        TMPC     RH  DRCT   SPED
    0.91  1013.10  0.15400E+02  86    12   1.00
    4.88  ///////  0.15900E+02  85   /// //////
    7.01  1012.20  0.15900E+02  85   355   0.72
   13.11  1011.50 ////////////  87   356   0.82
   17.99  1011.00  0.15900E+02  88   359   0.87
   20.12  1010.80  0.16000E+02 ///   /// //////
   24.09  1010.30  0.16100E+02  87    13   1.08
   28.05  1009.80  0.16100E+02  87    24   1.18
////////  1009.30  0.16200E+02  87    34   1.29
   38.11  ///////  0.16200E+02  87    43   1.29
   38.11  1008.50  0.16400E+02  87    48   1.39
```

Decode with: sample_decoder4.m

```
%  file = sample_decoder4.m
%
%  written by:  M. Jordan  02/27/2002
%
%  Purpose:  Decode the sounding 'sample_data4.txt', which contains ////'s
%        to indicate missing values.
%
%        1.  Each sounding data line must be read as a character string.
%        2.  Data values must be located within the string and
%            converted from character to number (str2num function).
%        3.  NaN = Missing values
%
%  Input:    None.  Filename is "hardwired".
%
%  Output:
%        stid     = station ID character string
%        datetime = date/time character string
%        lat      = latitude - real number
%        lon      = longitude - real number
%        elev     = station elevation - real number
%        iline    = number of sounding lines decoded
%        hght     = height (m)
%        pres     = pressure (mb)
%        tmpc     = temperature (C)
%        relh     = relative humidity (%)
%        drct     = wind direction (degrees)
%        sped     = wind speed (m/s)

%           ... open the data file, read only
fid_in = fopen('sample_data4.txt','r');

%           ... read each text header line
line1 = fgetl(fid_in);
line2 = fgetl(fid_in);
```

```
line3 = fgetl(fid_in);
line4 = fgetl(fid_in);

%  decode the information in the sounding header

stid = line1(9:11);            % station ID character string
datetime = line1(45:55);       % date/time character string
lat = str2num(line2(9:14));         % latitude - real number
lon = str2num(line2(28:34));   % longitude - real number
elev = str2num(line2(47:51));  % station elevation - real number

%           ... initialize a line counter
iline=0;

%           ... This WHILE LOOP to READ EACH DATA LINE is an
%        infinite loop, since "while 1" is always true.
%        The way to exit from the loop is with an if, break.
%        The "isstr" function will test if line is a character
%        string.  The if ~isstr(line) logic means if line is
%        NOT (~) a string.  So if ~isstr(line) is TRUE when
%        line is NOT a string, and is the end-of-file.
%        When end of file is reached, "break" exists the WHILE Loop.

while 1
%       ...  Read each line as a character string
   line=fgetl(fid_in);
%
%           ... Check for the effective "end-of-file"

   if ~isstr(line), break, end

%           ... Increment line counter.  This is the index for each
%        data array
   iline = iline+1;

%       ... IF logic to check for missing values, ////.  If not
%        '///', then convert string to number.

   if line(1:8) == '////////'
    hght(iline) = NaN;
   else
    hght(iline) = str2num(line(1:8));
   end
%
   if line(11:17) == '///////'
    pres(iline) = NaN;
   else
    pres(iline) = str2num(line(11:17));
   end
%
   if line(19:30) == '////////////'
    tmpc(iline) = NaN;
   else
    tmpc(iline) = str2num(line(19:30));
   end
%
   if line(32:34) == '///'
    relh(iline) = NaN;
   else
    relh(iline) = str2num(line(32:34));
   end
```

```
%
   if line(38:40) == '///'
    drct(iline) = NaN;
   else
    drct(iline) = str2num(line(38:40));
   end
%
   if line(42:47) == '//////'
    sped(iline) = NaN;
   else
    sped(iline) = str2num(line(42:47));
   end
%
end      % This is the end of the WHILE Loop

%   ... The IF, BREAK jumps to the command following this line.

fclose(fid_in);

sprintf('Number of lines in the input sounding: %5.0f',iline)

%      clear unnecessary variables
clear fid_in line line1 line2 line3 line4 ans

% ------------------------------------------------------------------------
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MATLAB SKILLS CHECKLIST

REFERENCES:

1. Text book: *Getting Started with MATLAB 5*, R. Pratap, 1999.
2. MR 2020 Course Notes

SKILLS:

1. Start MATLAB from the menu or UNIX shell (notes, p. 19)

2. MATLAB windows (section 1.6.1)

3. On-line help (sections 3.3, 1.6.2, 1.6.6 and Appendix A.2)

4. Input-output, format command, case sensitivity, use of the semicolon (section 1.6.3)

5. M-files and MAT files (section 1.6.4)

6. Workspace commands (section 1.6.6)

7. Use operating system commands from the command line: built-in commands (section 1.6.6) and use of the exclamation point before UNIX commands (notes, p.17).

8. Create and manipulate matrices (section 3.1)

9. Matrix operators (notes, p. 26, section 3.2.1, and Appendix A.6)

10. Array operators (notes, p. 31, section 3.2.1, and Appendix A.6)

11. Order of precedence (notes, p. 23)

12. Built-in functions (section 3.2.4 and Appendix A.7)

13. Loading and saving data (section 3.4.1)

14. Using the diary (section 3.4.2)

15. Punctuation marks and syntax (Appendix A.1)

16. General purpose commands (Appendix A.2)

17. Special variables and constants (Appendix A.3)

18. Ways to get data into MATLAB

    - manual entry (notes, p. 19, sections 2.1-2.2, section 3.1)
    - load binary ".mat" file (section 3.4)
    - load ASCII text file with columns of data, but without header lines, or header commented out with percent (%) in column 1 (see on-line help for "load" command).

19. "input" Command (notes, p. 36, section 4.3.5, p.94)

20. "disp" Command (notes, p. 38)

21. Search path and "path" Command (notes, p. 41)

22. Script M-files (notes, p. 43, section 4.1)

23. Relational and Logical Operators using Scalar Variables (notes, p. 49)

24. Compound Expressions using Relational and Logical Operators (notes, p. 49, class and lab examples)

25. "if" Statements (notes p. 53, section 4.3.4)

26. "for" Loops (notes, p. 60, section 4.3.4)

27. "while" Loops (notes p. 66, section 4.3.4)

28. "break" Command (notes p. 69, section 4.3.4, p. 92)

29. "error" Command (section 4.3.4, p. 93) ** not on test, but good to know

30. "return" Command (section 4.3.4, p. 94) ** not on test, but good to know

31. "keyboard" Command (section 4.3.5, p. 94) ** not on test, but good to know

32. "pause" Command (section 4.3.5, p. 96) ** not on test, but good to know

33. On-line help for script and function M-files (section 3.3, p. 64-69, section 4.3.1, p.88)

34. Function M-files (notes, p. 71, section 4.2, 4.2.1, 4.2.2, p. 80-88)

    - "feval" Command (section 4.2.2, p. 85) ** not on test, but useful
    - subfunctions (section 4.2.3, p. 87) ** not on test, but useful
    - parsed functions (section 4.2.4, p. 87) ** not on test, advanced topic
    - profiler (section 4.2.5, p. 87) ** not on test, advanced topic

35. "lookfor" Command (section 3.3, p. 64-69, section 4.3.1, p.88)

36. Continuation (…) Command (section 4.3.2, p. 89)

37. Understanding MATLAB Error Messages (chapter 7, p.197-202)

38. Punctuation marks and syntax (Appendix A.1, p. 213-214)

39. Operators and Logical Functions (Appendix A.6, p. 217)

40. The skills, commands and functions listed in the following tables:

Table 1. Basic plotting commands.

MATLAB Command or Topic	**Textbook Reference or Helpdesk: "Using MATLAB Graphics" topic**
plot plot(Temp,Ht,'r--*') plot(Temp,Ht,'b-',DewPt,Ht,'r—')	Sections 6.1, 6.1.1. and 6.1.5
hold on / hold off	Section 6.5 and on-line help
line	Section 6.5
subplot (multiple plots)	Section 6.2
figure figure(*number*)	See on-line help
xlabel	Section 6.1.2
ylabel	Section 6.1.2
title	Section 6.1.2
text	Section 6.1.2
gtext	Section 6.1.2
legend	Section 6.1.2
axis axis ij flips vertical axis	Section 6.1.3 -- see on-line help for newer syntax than in textbook
plotedit	Section 6.1.4 Helpdesk: Search for function "plotedit"
propedit	Sections 6.1.4 and 6.4
close close all	See on-line help
clf … clears current figure	See on-line help
Greek symbols in text strings ' \pi \Pi '	Helpdesk: Search for "Greek letters"

Table 2. Advanced plotting options: 2-D and 3-D plots.

MATLAB Command or Topic	**Textbook Reference or Helpdesk: "Using MATLAB Graphics" topic**
Specialized 2-D plots	Section 6.1.6, p. 154-159 help graph2d help specgraph
3-D plots	Section 6.3 help graph3d
plot3	Section 6.3
comet3	Section 6.3
view	Section 6.3.1
Mesh and surface plots Function: meshgrid	Section 6.3.3
Interpolated surface plots Function: griddata	Section 6.3.4

Table 3. Character String Manipulation

MATLAB Command or Topic	**Reference**
Character Strings • Rules for Character Strings • Character Strings as Row Vectors • Concatenation • Convert Numbers to Character Strings • Matrices of Character Strings • Individual Character Strings in a Matrix • Adding Strings to Existing Matrices • Examples using "for" loops	Notes, pp. 94-99 and Text Section 3.2.6
Character String Functions (built-in) • int2str • num2str • char • str2num • other functions	Notes, p. 94 and Text p. 62 • Notes, p. 95; On-line help • Notes, p. 95; On-line help • Notes, p. 96; On-line help • On-line help Text p. 62; Appendix 9, p. 220
"eval" function	Notes, p. 100; Text p. 62-63

Table 4. Searching arrays/matrices; replacing bad/missing data; plotting only good data.

MATLAB Command or Topic	Reference
Array indices: definitions and rules	Notes, p. 102
Matrix indices: definitions and rules	Notes, p. 103
"find" function	Notes, p. 104; Text p. 57
Logical arrays and matrices	Notes, p. 105
"logical" function	Notes, p. 105; on-line help
Logical and relational operators (for arrays)	Notes, p. 107; Text p. 54-57
Built-in logical functions	Text p. 57 and Appendix 6, p. 217
Search arrays and matrices	Notes, p. 108
"isnan" function	Notes, p. 108
Bad or missing data	Notes, p. 112
Data quality control (QC)	Notes, p. 112
Replacing bad or missing data values	Notes, p. 114
Excluding NaNs while plotting	Notes, p. 116

Table 5. Program design strategy and debugging.

MATLAB Command or Topic	Reference
Program design strategy	Notes, p. 118
Debugging suggestions	Notes, p. 120

Table 6. Formatted Output.

MATLAB Command or Topic	Reference
Writing to the screen • sprintf function • fprintf function	Notes, p. 121 Notes, p. 121 Notes, p. 128
Format strings	Notes, p. 124

Table 7. Statistical Functions.

MATLAB Command or Topic	Reference
Built-in statistical functions	Text Section 5.3, p. 123-124

Table 8. Reading ASCII Text Data Files

MATLAB Command or Topic	Reference
“fopen” function	Notes, p 131. On-line help.
“fclose” function	Notes, p 132. On-line help.
“fgetl” function	Notes, p 132. On-line help.
“fscanf” function	Notes, p 133. On-line help.